Verification of Low Level Software

Lecture Notes

Table of Contents

Handout, units 1 to 8 1
IOPS Software Requirements Document 1.0 46
IOPS Software Requirements Document 2.0 56
IOPS Architectural Design Document 1.0 66
Traceability Matrix for IOPS System Tests 76
Medcare Software Testplan Template 77
Advanced Coverage Metrics for Object-Oriented Software 98
Artikelreihe über Software-Codierungs-Standards 112
Risk based testing 148
Review Item Discrepancy (RID) Form 151
Annotated References 152

Copyright 2004 by Stephan Grünfelder,
Copyright 2001 by WEKA Zeitschriftenverlag,
Copyright 1999 by IPL Information Processing Ltd.

Verification of Low Level Software

Reviewing and Testing in the Software Life Cycle with an Emphasis on Embedded Systems

© Stephan Grünfelder

Participants & Expectations

- Who are you?
- What is your experience with software verification?
- What do you expect from the course?

© Stephan Grünfelder

Contents

Unit 1:

- Motivation
- Self-Assessment-Test
- Definitions
- V-Model, motivation of an early V&V
- Special considerations for embedded systems
- Review techniques
- How to review a software requirements document

© Stephan Grünfelder

Contents

Unit 1 (continued):

- Case studies of requirements
- *Review of the Software Requirements Document 1.0*
- *Walkthrough Review Meeting*

© Stephan Grünfelder

Contents

Unit 2:

- Using UML to analyse requirements & scenarios
- FMEA, Fault Trees
- *Review of the Software Requirements Document 2.0*
- *Technical Review Meeting*
- Test automation: advantages, drawbacks, costs
- *Writing a System Test Plan* (moderated team work)
- Traceability

© Stephan Grünfelder

Contents

Unit 2 (continued):

- Black box testing techniques
- System test design (principles, categories, examples)
- *System test design (exercise)*

© Stephan Grünfelder

Contents

Unit 3:

- *Discussion of the system test design*
- Organisational Approaches for Unit Testing
- Integration test plan
- What is good SW design, how does it affect testing?
- *Technical review of the Architectural Design Document*

Contents

Unit 4:

- Code metrics for Unit Testing and Integration Testing
- Baseline Testing
- *Tool based software metric calculation*
- Software coding standards
- Code review checklists
- *Code review*

Contents

Unit 5:

- *Code review meeting (software inspection)*
- Capabilities of static code checking tools
- *Application of a code checker to the reviewed software*
- *Follow-up review, metric calculation, and static check*
- Unit test design
- Host target testing
- *Getting started with Cantata and the TCD file format*

Contents

Unit 6:

- *Implementation of simplified unit tests* (moderated).
- A note on test metrics.
- Tools for automatic test case generation.
- *Design and implementation of unit tests.*

Contents

Unit 7:

- *Design and implementation of unit tests*
- Bug tracking
- Generating and using unit test error statistics
- A note on OO test metrics
- Unit test summary report
- *System test readiness review*

Contents

Unit 8:

- Generating and using system test error statistics
- *Running the designed system tests*
- System Test Summary Report
- Extreme Programming
- Risk based testing
- Conclusion
- *Feedback*

What is NOT taught here ...

- Formal specifications
- Static theorem proving
- Certification of safety critical software
- Verification of time domain issues in parallel systems
- Internet security testing

Unit 1

Let's start!

Today's contents

Unit 1:

- Motivation
- Self-Assessment-Test
- Definitions
- V-Model, motivation of an early V&V
- Special considerations for embedded systems
- Review techniques
- How to review a software requirements document

Today's contents

Unit 1 (continued):

- Case studies of requirements
- *Review of the Software Requirements Document 1.0*
- *Walkthrough Review Meeting*

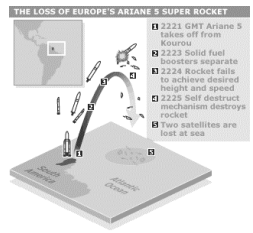
Motivation

Komplexere Software in Autos als Pannenursache Nummer eins

Die immer komplexere Software in Autos ist Pannenursache Nummer eins: Fehler in der Elektronik seien inzwischen für 55 Prozent der Ausfälle verantwortlich, räumten Automobilhersteller und Softwareexperten zu Beginn der Fachtagung «Informatik 2003» am Dienstag in Frankfurt ein. Hauptursache sei das mangelnde Zusammenspiel zwischen Systemen verschiedener Zulieferer.
Von 1998 bis 2001 sei die Zahl der Pannen, die durch fehlerhafte Software verursacht wurden, um 23 Prozent gestiegen. Andere Ursachen hätten nur um 3 Prozent zugenommen. Werde nicht schnellstens gegengesteuert, könnten softwarebedingte Pannen in zehn Jahren zwei Drittel aller Defekte ausmachen, rechnet die Gesellschaft für Informatik (GI) vor.
Für die Autoindustrie werde die Informatik immer wichtiger, bestätigt Ulrich Weinmann, Geschäftsführer der IWM Car IT GmbH. Der Markt für Automobilsoftware wird sich einer Studie der Unternehmensberatung Mercer zufolge bis 2010 auf 100 Milliarden Euro vervierfachen. Schon heute basierten 90 Prozent aller Innovationen im Auto auf Software oder Elektronik. «In einem Oberklasse-Wagen sind heute 70 oder 80 elektronische Steuergeräte drin», sagt Weinmann.
Dank Klimaanlagen, Navigationssystemen und High-Tech-Stereoanlagen sind Autos für die Fahrer angenehmer geworden. Antiblockiersysteme (ABS), Stabilisierungshilfen (ESP), Bremsassistenten und piepsende Einparkhilfen haben den Verkehr sicherer gemacht. Einer Untersuchung des Center Automotive Research in Gelsenkirchen zufolge hat sich die Zahl der Personenwagen in den vergangenen 30 Jahren mehr als verdoppelt, die Zahl der Unfalltoten aber sank um 70 Prozent. «Dies ist zu einem Gutteil auf Systeme wie ABS oder ESP zurückzuführen», glaubt die Gesellschaft für Informatik und beruft sich dabei auf Zahlen des Statistischen Bundesamtes und des ADAC.
Allerdings häufen sich auch die Systempannen. Prof. Heinrich Mayr, Präsident der GI, macht dafür eine fatale Koalition aus immer kürzeren Entwicklungszyklen und immer komplexeren IT-Systemen verantwortlich. «Bislang wird unterschätzt, dass der Entwicklungsprozess bei Software methodisch aufwendiger ist als in der Mechanik», sagt er. «Wir sollten von den Ingenieuren lernen - auch was Gründlichkeit angeht.» Er wünsche sich einheitliche Standards für alle Hersteller, Zulieferer und Programmierer.
Das würde auch bei einer Panne der Werkstatt die Arbeit erleichtern. Denn die ist nicht selten überfordert wegen der Vielzahl der Elektronikbausteile im Wagen. «Die Komplexität in der Automobil-Industrie ist dramatisch hoch», sagt Erich Nickel, Direktor für Automobil-Software bei IBM. In einem undurchschaubaren Systemverbund sei es extrem schwer, festzustellen, wo der Fehler liege. Die Werkstatt tausche dann gerne mal die Steuergeräte aus und schicke sie ein. Doch der Hersteller kann am Einzelgerät keinen Fehler entdecken. «Das ist ineffektiv und viel zu teuer.».

dpa 30.9.2003

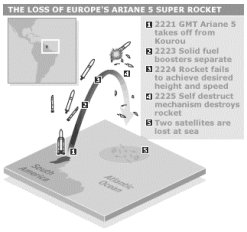
Motivation


THE LOSS OF EUROPE'S ARIANE 5 SUPER ROCKET
1 2221 GMT Ariane 5 takes off from Kourou
2 2223 Solid fuel boosters separate
3 2224 Rocket fails to achieve desired height and speed
4 2225 Self destruct mechanism destroys rocket
5 Two satellites are lost at sea
South America
Atlantic Ocean

Ariane 5 crashed after the maiden launch.

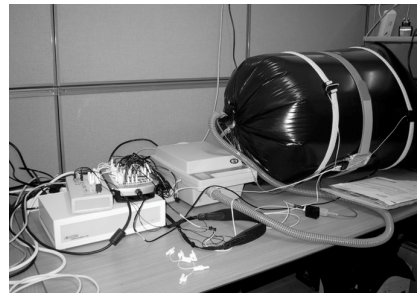
The proximate cause of the crash was an overflow error due to an attempt to convert a 64 bit floating point value into a 16 bit integer.

Motivation


THE LOSS OF EUROPE'S ARIANE 5 SUPER ROCKET
1 2221 GMT Ariane 5 takes off from Kourou
2 2223 Solid fuel boosters separate
3 2224 Rocket fails to achieve desired height and speed
4 2225 Self destruct mechanism destroys rocket
5 Two satellites are lost at sea
South America
Atlantic Ocean

This error occurred in code that was non-functional after liftoff, when the error occurred. Apparently an inertial platform from the Ariane 4 was used aboard the Ariane 5 without proper testing.

Testing Embedded Systems is Fun ...



A Self-Assessment Test

A program reads three integer values from the command line.

The three values are interpreted as representing the lengths of the sides of a triangle.

The program prints a message that states weather the triangle is scalene, isosceles, or equilateral (allgemeines Dreieck, gleichschenkelig, gleichseitig).

Evaluate your testing:

Solution:

Evaluate your testing:

Evaluate your testing:

Why does software contain bugs?

- Miscommunication or no communication
- Software complexity
- Changing requirements
- Time pressure
- Psychological issues (at meetings/general)
- Poorly documented or poorly designed legacy code
- Programming errors –
  programmers, like anyone else, can make mistakes

Definitions

Software verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

IEEE Standard Glossary of Software Engineering Terminology (1990)

Definitions

Software verification provides objective evidence that the design outputs of a particular phase of the software development life cycle meet all of the specified requirements for that phase. Software verification looks for consistency, completeness, and correctness of the software and its supporting documentation, as it is being developed, and provides support for a subsequent conclusion, that software is validated.

U.S. Department Of Health and Human Services, Food and Drug Administration, CDRH: General Principles of Software Validation; Final Guidance for Industry and FDA Staff.

Definitions

… *verification* activities include technical reviews, walkthroughs and software inspections, checking that design components are traceable to software requirements, testing, and audits.

European Space Agency Board for Software Standardisation and Control: ESA software engineering standards.

Definitions

Software validation is the confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use are fulfilled.
IEEE Standard Glossary of Software Engineering Terminology (1990)

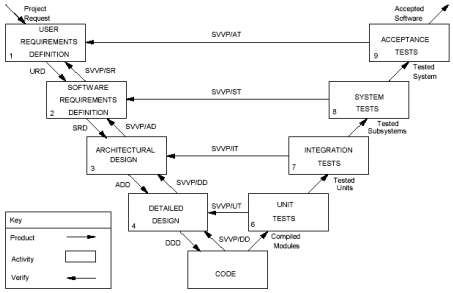
Verification asks the question, “Are we building the product right?”. *Validation* asks, “Are we building the right product?”
Everybody else

Definitions

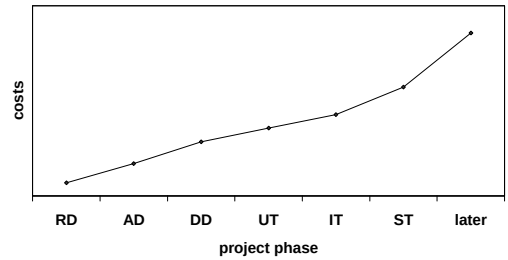
Testing is the process of analyzing a software item to detect the differences between existing and required conditions, and to evaluate the features of the software item.

Debugging means to detect, locate, and correct faults in a computer program.

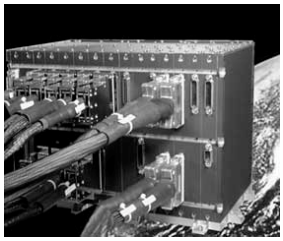
Software Lifecycle Model

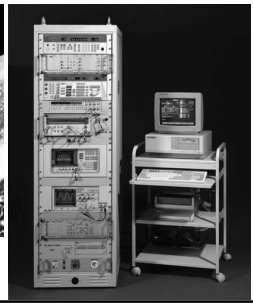


Costs of flaw repair vs. flaw detection time

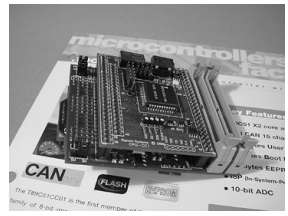


Special considerations for embedded systems



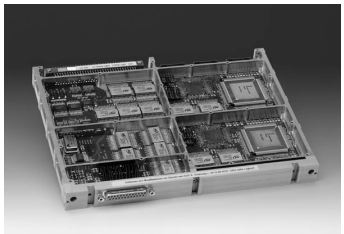


Special considerations for embedded systems



Integration testing must consider software/hardware interfaces. How can you access test and debug data in the fully integrated system?

Special considerations for embedded systems



How to run a unit test on the target hardware, when the test tool is designed for PC software, i.e. expects a stdio library and unlimited memory?

Special considerations for embedded systems

→ Testing must be planned carefully
(a sample test plan will be presented in unit 2).

→ Testing an embedded system can be a challenging project that may exceed the cost of developing the system by far.

Review techniques: walkthrough meetings

Preparation

A review leader nominates the team.

The moderator or author distributes the review items when the author decides that they are ready for walkthrough. Members should examine the review items prior to the meeting. Concerns should be noted so that they can be raised at the appropriate point in the walkthrough meeting.

Review techniques: walkthrough meetings

Review Meeting

The author provides an overview of the review items.

A general discussion follows, during which issues of the structure, function and scope of the review items should be raised. The author then steps through the review items. Members raise issues about specific points as they are reached in the walkthrough. As the walkthrough proceeds, errors, suggested changes and improvements are noted by the leader.

Review techniques: walkthrough meetings

Output

- a list of the members
- a list of changes and defects noted during the walkthrough
- a list of actions, with persons responsible identified and expected dates for completion defined
- recommendations made by the walkthrough team on how to remedy
- defects and dispose of unresolved issues (e.g. further meetings)

Review techniques: software inspection

Output

- a list of the members
- a list of changes and defects noted during the software inspection
- a list of actions, with persons responsible identified and expected dates for completion defined
- ~~recommendations made by the walkthrough team on how to remedy~~
- defects and dispose of unresolved issues (e.g. further meetings)

Review techniques: technical teview

Preparation

The leader creates the agenda and distributes it, with the statements of objectives, review items, specifications, plans, standards and guidelines to the review team. Members then examine the review items. Each problem is recorded on the RID form. A RID form should record only one problem, or one group of related problems. Members then pass their RIDs to a secretary, who numbers each RID uniquely and forwards it to the author for comment. Authors add their responses on the RID form and then return the it to the secretary. RIDs are categorised as major, minor, or editorial.

Review techniques: technical teview

Review Meeting

Major RIDs are discussed, followed by the minor and editorial RIDs. The outcome of the discussion of any defects should be noted by the review leader in a review decision box on the RID form. This may be one of CLOSE, UPDATE, ACTION or REJECT.

Review techniques: technical teview

Output:

- the list of the reviewers and meeting participants
- a table of RIDs organised according to category, with dispositions marked;
- a list of actions, with persons responsible identified and expected dates for completion defined;
- the meeting conclusion

Review techniques: audit

The objective of an audit is to verify that software products and processes comply with standards, guidelines, specifications and procedures.

The audit process is carried out by an audit team who interview the development team, examine review items, report errors and recommend solutions.

Advantages of Reviews

- **Reviews improve the self-awareness of developers.**
  Reviews uncover not only weaknesses of product (and processes) but also of developers. They promote a realistic self-awareness and this means nearly always a more modest one.
- **Reviews level out the individual scales.**
  Discussion causes an increase of harmony in standards if the reviews are carried out in a team. A new culture emerges (egoless programming).

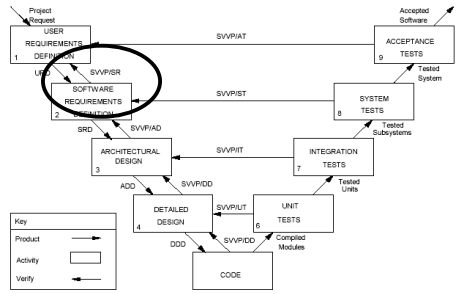
Advantages of Reviews

- **Reviews point out dead ends and lead out of them.**
  Dialog produces helpful pressure to clear up and explain the individual position. Therefore, the developer often becomes aware of defects himself which he had overlooked working all alone.
- **Reviews can be applied early in software development process.**
  Reviews can be applied before executable code is available to test it. So the latency period of faults in the software development can be shortened significantly.

Problems with Reviews

- **Reviews require effort.**
  The preparation and the meeting itself need time and qualified meeting participants.
- **Reviews can threaten the individual.**
  A developer can easily be turned from author to an examined person (prevention by the moderator). If this effect is ignored, there is great resistance to reviews.

How to review a SRD



How to review a SRD

Technical feasibility

How to review a SRD

Stylistic Checklist:

- Write short sentences.
- Clearly define what is part of the requirement and what is not (shall, should, delimiters).
- Avoid passive voice.
- Avoid multiple statements in one sentence: “the software checks if the input condition holds and the state matches or matched in the previous query” .
- Use tables, lists, and enumerations to state larger amounts of requirement text.

How to review a SRD

Stylistic Checklist:

- Use diagrams to illustrate complex requirements.
- Avoid common speech and undefined abbreviations.
- Keep requirements descriptions short .
- Avoid reduncancies, make unambiguous cross-references only.
- Avoid anticipating the software design.
- Write testable requirements.

How to review a SRD

Contents Checklist [RUP]:

- Functional Requirements
- Usability Requirements
- Requirements on Reliability and Accuracy
- Performance Requirements
- Supportability (coding standard, maintenance access)
- Design Constraints (language, code re-use, ...)
- Interfaces (hardware, software, user)
- Standards, legal constraints

How to review a SRD

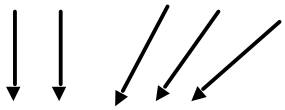
Consistency Checklist [PSS-05]:

Danger if ...

- different terms are used for the same thing
- the same term is used for different things
- incompatible actions are required at the same time
- actions are required in the wrong order

Example Requirement

R/SRD/Strt/10-2: If the memory self test fails, the DSP shall clear the error register within 50ms after the start of the self test. #



R/SRD/Strt/10-2: If the memory self test fails, the DSP shall clear the error register within 50ms after the start of the self test.. #

Bad requirement 1

R/SRD/Strt/10-2: Bit PWR of the Start Release Message limits the activaton time of the starter motor. After ECU init this release bit is set. The bit will be reset after the starter has been activated for a calibratable time and in case of ignition off. After a recovery and in the initialisation the release bit has to be set. #

Bad requirement 2

R/SRD/5.2/12-1: To optimise gear shifting in the automatic transmission, the state of the compressor shall be frozen for the time of the gear shift. #

Bad requirement 3

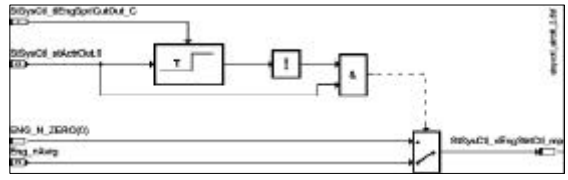
R/SRD/Dia/15-1: The diagnosis can be turned off in certain vehicle conditions (message from the engine coordinator). #

Bad requirement 4

R/SRD/AC/35-2: If the A/C pressure falls below a minimum hysteresis threshold, an instant shut off condition shall be fulfilled. An instant shut off shall also be fulfilled when the A/C pressure is over a maximum hysteresis threshold. When using an A/C pressure cut off switch (should be named ACCtl_swtCutoff_C), an instant shut off shall also be fulfilled when the cutoff switch label (ACCD_stACCutoff) is not set. #

Bad requirement 5

R/SRD/StrtCtl/35-1: For the engine start, instead of the average engine speed, StSysCtl_nEngStrtCtl_mp shall be used as shown in the figure below. #



Bad requirement 6

3.5 Quantity Governing - Exceptions CCTL24

Validity [P444 HN 2.0.0, ...]

Desc Since the CCTL - Governor works as a quantity Governor, the torque coming as input to CCTL shall be converted into quantity (Injection Mass). In the same way the quantity needs to be conveted into torque to get an interface in the torque path. The Chrysler - 395 CCTL has fuel density value = 1 and hence injection mass and quantity are same. Hence PRP_NORM_CONV_FAC shall have Injection mass instaed of Injection quantity.

Ratio

Check

Com

```
systemkonstante  PRP_NORM_CONV_FAC
{
   wertp      (TRQPRPHIGH_RES / FACT1_RES / INJ_MASS_RES);  /* [Nm/(mg/hub)] */
}
```


Bad requirement 7

R/SRD/CrCtl/15-2: When the last state of the panel was in NEUTRAL and the current position is COAST/SET and the Governor is not ACTIVE and no shut-off conditions present, the CCTL state maching mode shall transit to MODE SET. #

Introduction to the Virtual Project

MyConsultant

TheCompany Ltd

Intelligent Oil Pressure Switch

Plan reviews
Establish test plan
Testing (Unit, Integration, System)

Review of the IOPS SRD 1.0

- Technical feasibility
- Consistency
- Style:
  - Write short sentences.
  - Clearly define what is part of the requirement and what is not (shall, should, delimiters).
  - Avoid passive voice.
  - Avoid multiple statements in one sentence: "the software checks if the input condition holds and the state matches or matched in the previous query" .
  - Use tables, lists, and enumerations to state larger amounts of requirement text.
  - Use diagrams to illustrate complex requirements.
  - Avoid common speech and undefined abbreviations.
  - Keep requirements descriptions short .
  - Avoid reduncancies, make unambiguous cross-references only.
  - Avoid anticipating the software design.
  - Write testable requirements.
- Content: Functional Requirements, Usability Requirements, Requirements on Reliability and Accuracy, Performance Requirements, Supportability (coding std, maintainance access), Design Constraints (language, code re-use, ...), Interfaces (hardware, software, user), Standards, legal constraints

Review Meeting of the IOPS SRD 1.0

... what kind of review meeting?

Verification of Low Level Software

Unit 2

Contents

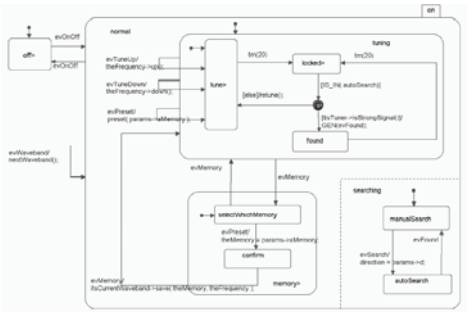
- Using UML to analyse requirements & scenarios
- FMEA, Fault Trees
- *Review of the Software Requirements Document 2.0*
- *Technical Review Meeting*
- Test automation: advantages, drawbacks, costs
- *Writing a System Test Plan* (moderated team work)
- Traceability

Contents

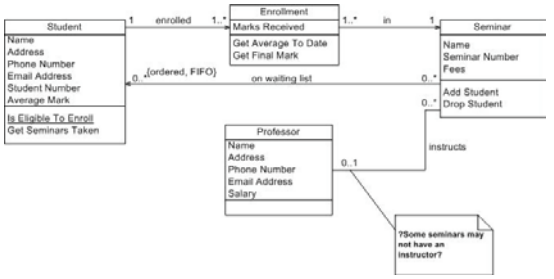
Unit 2 (continued):

- Black box testing techniques
- System test design (principles, categories, examples)
- *System test design (exercise)*

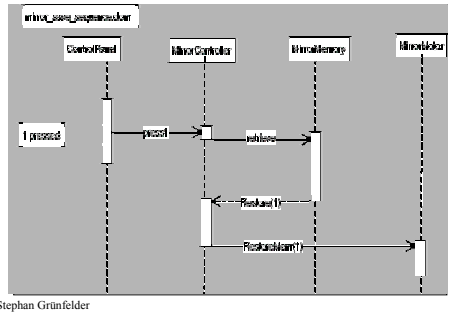
UML as Analysis Tool



UML as Analysis Tool



UML as Analysis Tool



FMEA

Failure Mode an Effect Analysis

Part & Function	Potential Failure Mode	Potential Effects of Failure	S E V	Potential Cause(s) of Failure	O C C	D E T	R P N

FMEA: serverity ranking SEV

Ranking	Effect	Criteria: Severity of Effect
1	None	No effect
2	Very Minor	Very minor effect on product or system performance.
3	Minor	Minor effect on product or system performance.
4	Low	Small effect on product performance. The product does not require repair.
5	Moderate	Moderate effect on product performance. The product requires repair.
6	Significant	Product performance is degraded. Comfort or convenience functions may not operate.
7	Major	Product performance is severely affected but functions. The system may not be operable.
8	Extreme	Product is inoperable with loss of primary function. The system is inoperable.
9	Serious	Failure involves hazardous outcomes and / or noncompliance with govt. regulations or standards.
10	Hazardous	Failure is hazardous, and occurs without warning. It suspends operation of the system an/or involves noncompliance with govt. regulations

Source: Colorado School of Mines, Division of Engineering

FMEA: likelyhood of occurance OCC

Ranking	Possible Failure Rates	Probability of Failure
1	$\leq 1 \times 10^{-6}$	Nearly Impossible
2	1×10^{-5}	Remote
3	1×10^{-4}	Low
4	4×10^{-4}	Relatively Low
5	2×10^{-3}	Moderate
6	1×10^{-2}	Moderately High
7	4×10^{-2}	High
8	0.2	Repeated Failures
9	0.33	Very High
10	≥ 0.55	Extremely High: Failure Almost Inevitable

Source: Colorado School of Mines, Division of Engineering

FMEA: likelyhood of detection DET

Ranking	Detection Probability
1	Almost Certain Detection
2	Very High Chance of Detection
3	High Probability of Detection
4	Moderately High Chance of Detection
5	Moderate Chance of Detection
6	Low Probability of Detection
7	Very Low Probability of Detection
8	Remote Chance of Detection
9	Very Remote Chance of Detection
10	Absolute Uncertainty – No Control

Source: Colorado School of Mines, Division of Engineering

FMEA

RPN = Risk Priority Number

RPN = OCC • SEV • DET.

A 1000 rating implies a certain failure that is very likely, hazardous, and impossible to detect.

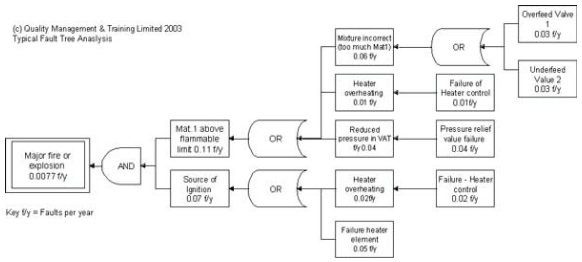
A 1 rating is a failure that is highly unlikely, obvious, and unimportant.

Rating below 30 are reasonable for typical applications.

FMEA Example

Part & Function	Potential Failure Mode	Potential Effects of Failure	S E V	Potential Cause(s) of Failure	O C C	D E T	R P N
IDMA Interface, transfer of EEC data from DSP to MPC	MPC reads other data than DSP writes	EEC displayed is incorrect	8	Static discharge when touching the device during application of the electrodes	5	6	240

FTA - Fault Tree Analysis


(c) Quality Management & Training Limited 2003
Typical Fault Tree Analysis
Major fire or explosion 0.0077 f/y
AND
Mat.1 above flammable limit 0.11 f/y
Source of Ignition 0.07 f/y
OR
OR
Mixture incorrect (too much Mat1) 0.06 f/y
Heater overheating 0.01 f/y
Reduced pressure in VAT f/y 0.04
Heater overheating 0.026/y
Failure heater element 0.05 f/y
OR
Failure of Heater control 0.016/y
Pressure relief value failure 0.04 f/y
Failure - Heater control 0.02 f/y
Overfeed Valve 1 0.03 f/y
Underfeed Valve 2 0.03 f/y
Key f/y = Faults per year

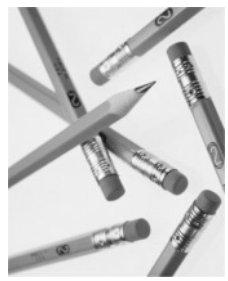
FMEA, FTA for IOPS

- FMEA
- FTA with following top level scenarios: (1) the IOPS is alive but does not communicate via the PP-bus, (2) the IOPS uses undefined or wrong debounce parameters.

Use UML sequence diagrams for analysis!

→ IOPS SRD 1.0 exhibits a system design flaw

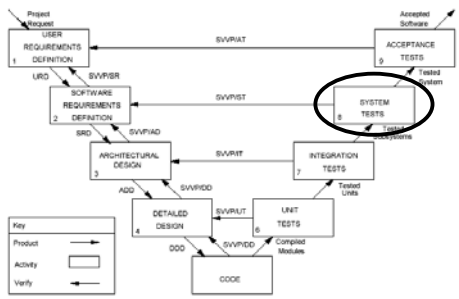
Review of IOPS SRD 2.0 using RID forms



Review Meeting

... what type of review meeting?

System Test Automation



System Test Automation

- Stimulation of the inputs of the system.
- Measurement of the outputs and comparison with expected outputs.
- Creation of useful test logs.

Advantages of Test Automation

- Repeatability (GUI example, GUI problem)
  - No forgetting of a step in a test case.
  - Clear proof of an error in case it is intermittend.
- Cost benefits for repetitive tests.
- Programming is fun.

Disadvantages of Test Automation

- Automating a test is 5 to 10 times the effort of manual testing.
- An automated test only checks what it is programmed for and cannot detect obvious unplausibilities, if it is not programmed to check for it.
- Sometimes impossible.
- Costs of the test equipment.
- GUI.

Advantages of Manual Testing

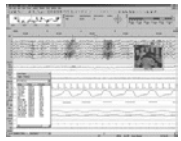
- Since a manual testing is faster to implement than an automated test, you can do more testing in the same time (unless you need to repeat the testing a couple of times).
- A human tester can find errors she wasn't even testing for.
- Costs of test equipment are less.
- Humans have no problems with GUI.

Disadvantages of Manual Testing

- Boredom.
- Prone to errors.
- Lacking repetitiveness.
- Sometimes impossible (e.g. volume testing).

Half Automatic Tests

- Operator promted for manual interaction by test script
- An operator must interpret data generated in an otherwise automatic test



Test Hooks

- Generation of test data in the test object
- Access to internal test data for authorised users

Automation in IOPS Software System Testing?

In the lab you have

- a PC-based PP-bus node (with Windows API) with interactive test software that produces log files with time stamps (resultion 3/100s) on a terminal window.
- A digital I/O card with C/C++ and LabView drivers.
- An IEEE bus PCI card with C/C++ drivers.
- A digital storage oscilloscope programmable via IEEE bus.
- A function generator, programmable via IEEE bus.

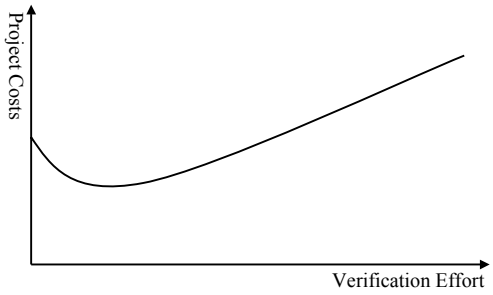
... to what extent will you automate software system testing?

Psychology

A programmer should avoid attempting to test his or her own program.

Testing is a destructive process: you try to break the software, try to find weaknesses, try to reveal bugs. It is extremely difficult, after a programmer has been constructive while designing and coding a program, to suddenly change his or her perspective and attempt to form a completely destructive frame of mind toward the program. Hence, most programmers cannot effictively test their own software because they cannot bring themselves to form the mental attitutde of exposing errors. [Myers78]

Economy



Write a System Test Plan

Testing is a project. Like in development projects the chance of success increases if you plan the work and keep working on the plan.

Define software release criteria. Make release criteria SMART:

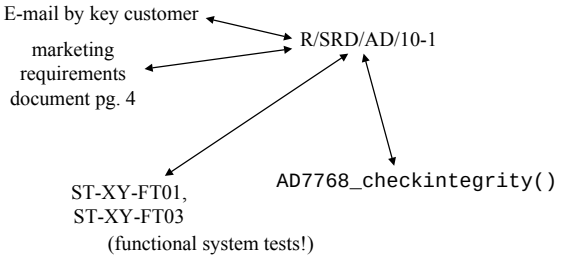
- Specific for the project
- Measurable to evaluate the software against them
- Attainable
- Relevant
- Trackable

Write the IOPS System Test Plan

Let's fill out the management sections of the Software Test Plan Template that MyConsultant wrote for the company Medcare.

Let's see of much use it is to IOPS.

Traceability



Functional System Test Design Techniques

- Equivalence partitioning
- Boundary value analysis
- (State transition testing)
- Error guessing

Black Box

Equivalence Partitioning

Partition the input (output) domain of a program into a finite number of equivalence classes such that one can reasonably assume that the test of a representative value of each class is equivalent to a test of any other value.

Examples:

- Triangle self assessment test of unit 1.
- *Test a program that returns the absolute value of an integer!*
- *Test a program that adds two integer values!*

Boundary Value Analysis

Experience shows that test cases that explore boundary conditions have a higher payoff than test cases that do not. Boundary conditions are those situations directly on,above, and beneath the edges of input or output equivalence classes.

Extend the examples you did before:

- *Triangle self assessment test of unit 1.*
- *Test a program that returns the absolute value of an integer!*
- *Test a program that adds two integer values!*

Error Guessing

Never just test the function of the requirements. Think what error could be in the software.

Caution

Think of errors in the test environment (positive/negative testing).

System Test Categories

- Function Testing
- Data Integrity Testing
- Business Cycle Testing
- Performance Profiling
- Storage Testing
- Load Testing
- Stress Testing
- Volume Testing
- Security and Access Control Testing
- Failover and Recovery Testing
- Configuration Testing
- Installation Testing
- Documentation Testing

Function Test Design Guidelines

- Use the described black box techniques
- Make the tests hard to pass
- Do not test things which are not required
  (a mistake that applies mainly to automated tests)
- Make the tests orthogonal
  (i.e. when you have to change one requirement, a minimal set of tests must be changed)
- Give clear orders in the textual description of the design, use active voice.

Example Function Test

R/SRD/XY/1-1: If the vessel temperature is higher than 300K, the DSP software shall set the bit OVRTMP of the CAN bus message VESERR. #

R/SRD/XY/2-1: If the vessel temperature is lower than 290K, the DSP software shall clear the bit OVRTMP of the CAN bus message VESERR. #

ST-XY-FT01: Set the system up such that VESERR is zero. Set the vessel temperature to the minimum possible. Wait 10 seconds.
(1) Verify that VESERR is zero.

Set temperature to 299K, wait 10 seconds. (2) Verify that VESERR is zero.

Set the vessel temperature to 301K, wait 1 second. Verify that bit OVRTMP of message VESERR is set but all other bits are zero.

Set temperature to 289K, wait 1 second. (3) Verify that VESERR is zero.

Set the vessel temperature to the maximum possible, wait 1 second. (4) Verify that bit OVRTMP of message VESERR is set but all other bits are zero.

Bad Function Test 1

Test objective: Check the oil pressure warning lamp.

Technique: Set the debounce times OPSCD_DebNeg_C and OPSCD_DebPos_C to 1s. Connect pin K14 with ground to simulate high oil pressure.

Wait one second. Verify that OPSCD_Lamp_mp is one.

Disconnect K14 and ground to simulae low oil pressure.

Wait one second. Verify that OPSCD_Lamp_mp is zero.

Bad Function Test 2

Test Objective: Verify that the speed fans are only switched on, when the fan ratio FanCtl_rFanPwr is between a maximum Fan_rFanOn_C and minimum Fan_rFanOff_C. Also verify if there is no drive away roar (Fan_stDrvAway is zero), it must be possible to set a limit.

Technique: First of all the Fan_stDrvAway_mp has to be zero. Only if this measurement point is set to zero, the FanCtl_rFanPwr can be controlled by the hysteresis. The engine speed has to be set to a high value (i.e. 5000 rpm). First set the Fan_rFanOff_C and the Fan_rFanOn_C to zero. The speed fans have to start running immediately.

[...]

IOPS System Test Design

Req#	Req Short Text	Tested in
R/SRD/INIT/10	Self Test	ST-IOPS-FT01
R/SRD/INIT/20	Self Test Fail Silence	ST-IOPS-FT01
R/SRD/PP/05	Ready Message	ST-IOPS-FT01
R/SRD/PP/20	PP-bus Performance	ST-IOPS-FT02
R/SRD/PP/30	Reply Message Format	ST-IOPS-FT02
R/SRD/PP/40	SetPosDebTime Cmd	ST-IOPS-FT03
R/SRD/PP/50	SetNegDebTime Cmd	ST-IOPS-FT03
R/SRD/PP/60	SetRepIntv Cmd	ST-IOPS-FT02
R/SRD/PP/70	Start Command	ST-IOPS-FT02
R/SRD/PP/80	Report Message Format	ST-IOPS-FT02
R/SRD/SI/10	Switch Position Sampling Rate	ST-IOPS-FT03
R/SRD/SI/30	Debouncing Algorithm	ST-IOPS-FT04, ST-IOPS-FT03
R/SRD/RA/10	Missing Parameters	ST-IOPS-FT05
R/SRD/RA/20	PP-bus Error	ST-IOPS-FT05
R/SRD/RA/30	Watchdog Kicking	ST-IOPS-FT05
R/SRD/SB/10	Maximum CPU Load	ST-IOPS-FT06
R/SRD/SB/20	CPU Idle Time	ST-IOPS-FT07
R/SRD/SB/30	Memory Footprint	ST-IOPS-FT08

Unit 3

Verification of Low Level Software

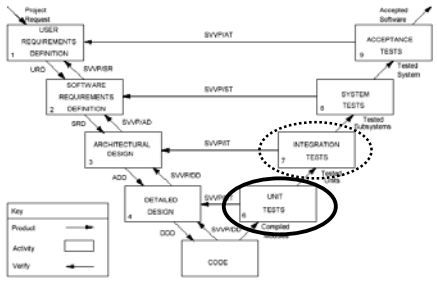
Contents

- *Discussion of the system test design*
- Organisational Approaches for Unit Testing
- Integration Test Planning
- What is good SW design, how does it affect testing?
- *Technical review of the Architectural Design Document*

Discussion of the IOPS System Test Design

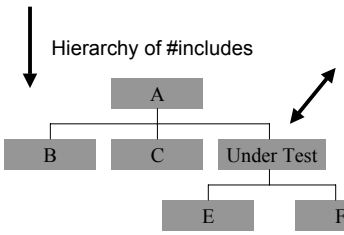
Annotaded IOPS SW Test Plan.pdf

Organisational Approaches for Unit Testing

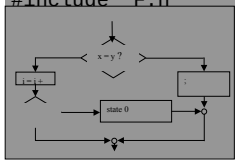


Organisational Approaches for Unit Testing

Hierarchy of #includes




```
#include "E.h"
#include "F.h"
```

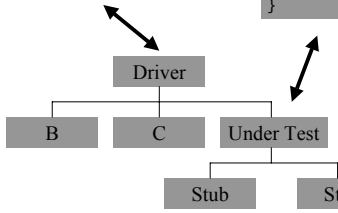


Organisational Approaches for Unit Testing

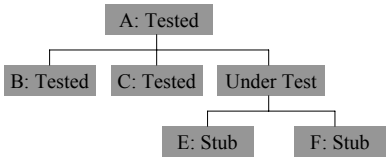
```
assert(cos(PI/2) == 0);
assert(cos(0) == 1);
assert ...
```

```
float cos(float x)
{
  return sin(PI/2-x);
}
```

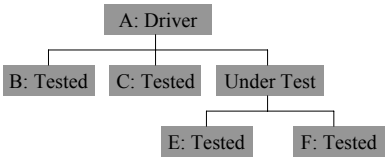
```
float sin(float x)
{
  static int call = 0;
  switch (call)
  {
    if (call == 0)
    {
      assert(x == 0);
      ...
```



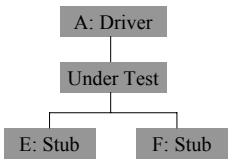
Top Down Testing



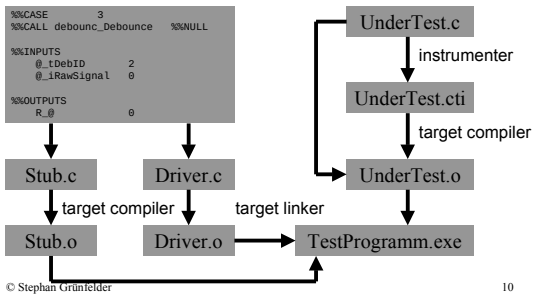
Bottom Up Testing



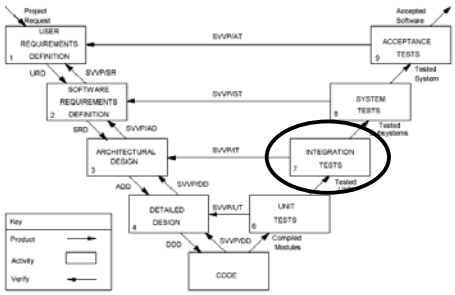
Isolation Testing



Isolation testing tool support



Integration Testing



Integration Testing

Integration tests verify that the software components work correctly with the rest of the system, and as specified in the architectural design.

Consequently integration tests checks that

- all data exchanged across an interface agrees with the data structure specifications and
- confirm that all the control flows in the ADD have been implemented.

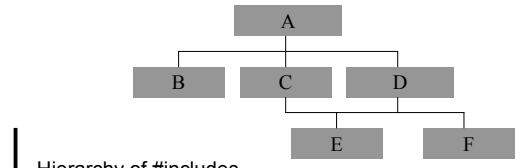
Integration Testing

Possible Approaches:

- Buttom up unit testing as shown before.
- McCabe's structured integration testing (unit 4).
- Set break points at interfaces and inspect data and control flow.
- Perform system tests with an instrumented code that reveals Call Pair Coverage.
- Perform system tests and analyse the Call Pair Coverage manually.

How does SW design influence testing?

Example 1:

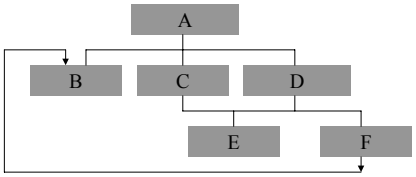


Hierarchy of #includes

→ no problem

How does SW design influence testing?

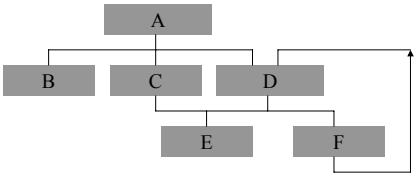
Example 2:



→ no problem

How does SW design influence testing?

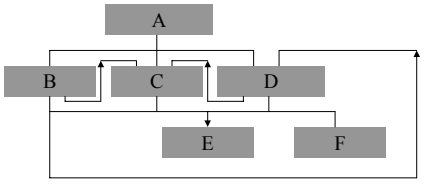
Example 3:



Problem → no loops!

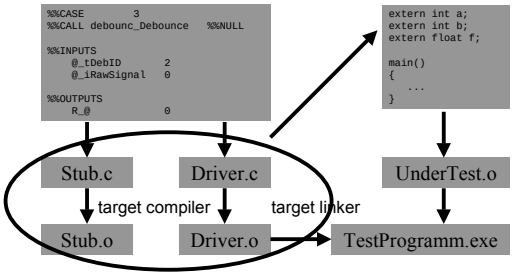
How does SW design influence testing?

Example 4:

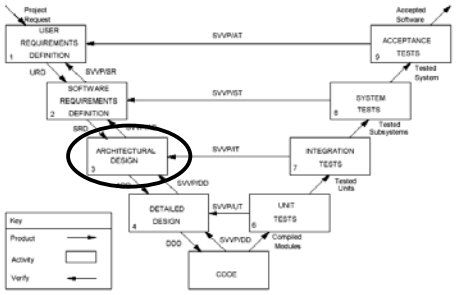


lacking cohesion → stubbing hardly manageable!

How about global variables?



Review of an ADD



Review of an Architectural Design Document

- Modularity.
- No cycles in the module dependencies.
- Cohesion.
- Data/Information hiding.
- Can all requirements indeed be implemented in the design elements given in the traceability matrix? Are requirements forgotten?
- Style: use the “same language“ as the SRD.
- Visualisation: use (standardised) figures to capture your design (UML, HOOD, ...)
- Clarity and maintainability of the design.

Review of the IOPS ADD

- Modularity.
- No cycles in the module dependencies.
- Cohesion.
- Data/Information hiding.
- Can all requirements indeed be implemented in the design elements given in the traceability matrix? Are requirements forgotten?
- Clarity and maintainability of the design.
- Style: use the “same language“ as the SRD.
- Visualisation: use (standardised) figures to capture your design (UML, HOOD, ...)
- Clarity and maintainability of the design.

Consistency concern

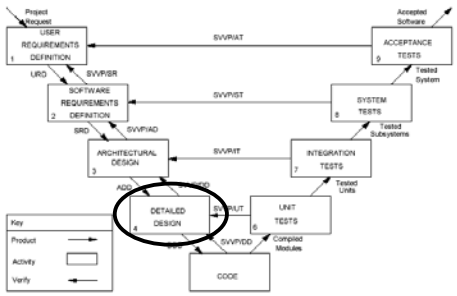
Q: How can you guarantee (or at least check) consistency between the description in the design document and the actual code?

A: Use a design tool with a code generator (and in the best case a reverse engineering interface).

Select an integration test strategy for the IOPS SW

- Buttom up unit testing.
- McCabe's structured integration testing (unit 4).
- Set break points at interfaces and inspect data and control flow.
- Perform system tests with an instrumented code that reveals Call Pair Coverage.
- Perform system tests and analyse the Call Pair Coverage manually.

Detailed Design Review



Unit 4

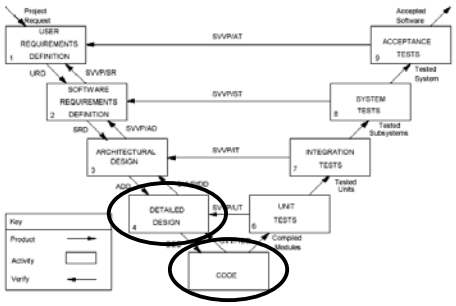
Verification of Low Level Software

Contents

Unit 4:

- Code metrics for Unit Testing and Integration Testing
- Baseline Testing
- *Tool based software metric calculation*
- Software coding standards
- Code review checklists
- *Code review*

Code metrics



Code metrics

The testability of software is strongly related to its complexity. Given that program statements are linked to each other by sequence, selection (i.e. branches or conditions) and iteration (i.e. loops), McCabe defined a metric that gives a complexity of 1 for a simple sequence of statements with no branches. Only one test is required to execute every statement in the sequence. Each branch added to a module increases the complexity by one, and requires an extra test to cover it.

Code metrics: cyclomatic complexity

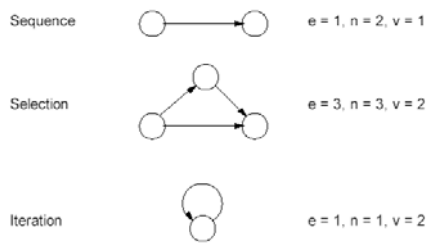
A control graph represents the control flow in a module. Control graphs are simplified flowcharts. Blocks of statements with sequential logic are represented as nodes in the graph. Branches between blocks of statements (called edges) are represented as arrows connecting the nodes. **McCabe** defines the **cyclomatic complexity** v of a control graph as:

$$v = e - n + 2$$

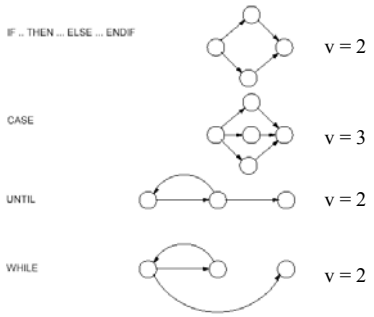
where:

- e is the number of edges;
- n is the number of nodes.

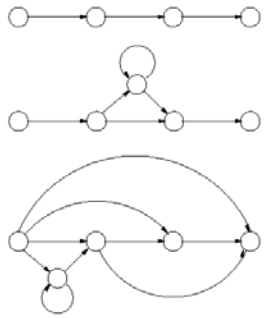
Cyclomatic complexity



Cyclomatic complexity

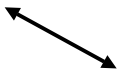


Cyclomatic complexity



Inaccuracy of the cyclomatic complexity

```
if ((a == 0) && (b > 2))
{
   printf("test\n");
}
```



```
if (a == 0)
{
   if (b > 2)
   {
      printf("test\n");
   }
}
```

© Stephan Grünfelder

Total cyclomatic complexity

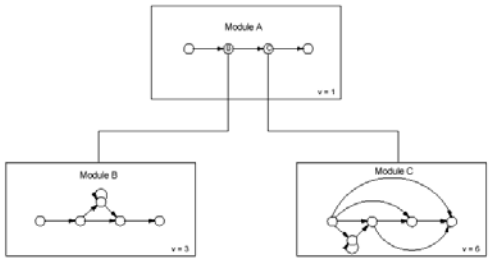
The total cyclomatic complexity of a program can be obtained by summing the cyclomatic complexities of the constituent modules. The full cyclomatic complexity formula given by McCabe is:

$$v = e - n + 2p$$

where p is the number of modules.

© Stephan Grünfelder

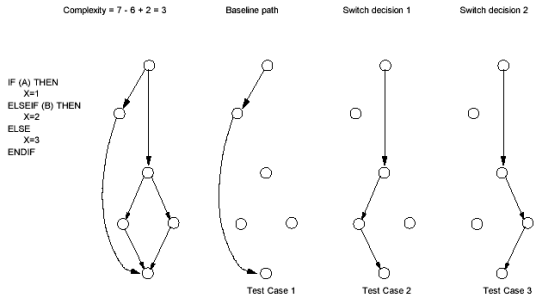
Total cyclomatic complexity



e = 18, n = 14, v = 18 - 14 + (2 x 3) = 10

© Stephan Grünfelder

Baseline testing method



© Stephan Grünfelder

McCabe's integration complexity

Module design complexity, denoted as iv by McCabe, measures the individual effect of a module upon the program design. The module design complexity is evaluated by marking the nodes of the control graph of the module that contain calls to external modules. Then the graph is 'reduced' according to the rules listed below:

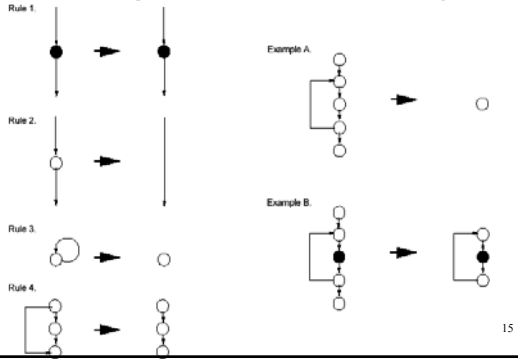
1. marked nodes cannot be removed;
2. unmarked nodes that contain no decisions are removed;
3. edges that return control to the start of a loop that only contains unmarked nodes are removed;
4. edges that branch from the start of a case statement to the end are removed if none of the other cases contain marked nodes.

McCabe's integration complexity

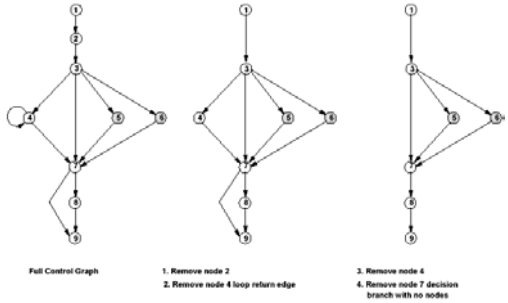
The module design complexity is the cyclomatic complexity of the reduced graph.

In summary, module design complexity ignores paths covered in module testing that do not result in calls to external modules. The remaining paths are needed to test module interfaces.

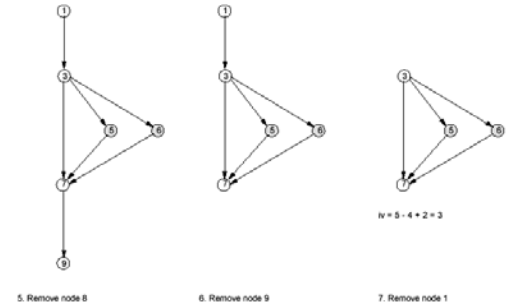
Integration complexity



Integration complexity


Full Control Graph
1. Remove node 2
2. Remove node 4 loop return edge
3. Remove node 4
4. Remove node 7 decision branch with no nodes

Integration complexity



Structured integration testing

The integration complexity, denoted as S_1 by McCabe, of an assembly of modules, counts the number of paths through the control flow. The design complexity $S_0 = \Sigma(vi)$. The integration complexity of an assembly of N modules is given by the formula:

$$S_1 = S_0 - N + 1$$

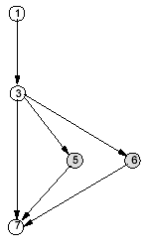
(minimum number of integration test cases of a project)

$\rightarrow$ when integrating a new module execute the control flow of the reduced graph.

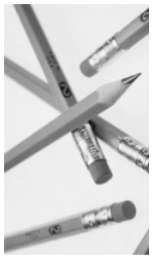
Review of the integration testing strategies (unit 3)

- Buttom up unit testing as shown in unit 3.
- McCabe's structured integration testing
- Set break points at interfaces and inspect data and control flow.
- Perform system tests with an instrumented code that reveals Call Pair Coverage.
- Perform system tests and analyse the Call Pair Coverage manually.

Integration testing depth



What kind of errors pass integration testing with 100% Call Pair Coverage but are found with McCabe's Structured Integration Testing?



Other software metrics

- Enhancements of cyclomatic complexity that regard the mentioned drawback.
- Number of goto:s.
- Number of exit points of functions.
- Lines of code.
- Ratio comments/statements.
- Maximum nesting level.

Use of software metrics for verification

- Effort estimation of unit and integration tests.
- Rejection of designs that exceed metric thresholds, i.e.
  - functions with v > 10
  - goto statements
  - more than one exit point per function
  - less than 20% comment lines
  - nesting level higher than 5

Tool based metric calculation

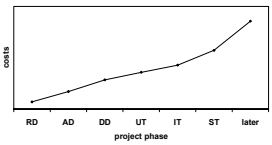
`metric` <source file name without extension>

Decide upon the metrics which source files are ready for code review (results in <src>.ctl).

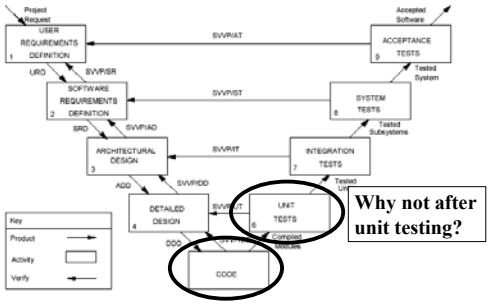
Functions with v ≤ 10, no goto:s, single exit point per function, comment lines/code lines > 20%, maximum nesting level of 5.

Code reviews

Code reviews can be partially automated by software tools (unit 5). However, a manual review is indispensable and the cheapest and most effective way to find software errors.



Code reviews



Why not after unit testing?

Code reviews

Each programmer may have his/her personal programming style and personal ideas of good or bad program code.

The need of code reviews means that programmers need to work as a team but different programming styles make the teamwork difficult.

→ **Software coding standards**

Software coding standards

help to

- improve the portability of the code,
- assisting in writing robust code,
- improve the readability of the code,
- and consequently ease co-operation in a team of programmers and reviewers.

Software coding standards

- Lexical standards (Hungarian notation, file names, function names, acronym standards, ...).
- Code layout rules.
- Comment rules (what has to be commented, where, to which extent).
- Design rules (goto:s, number of exit points, ...).
- Software build rules (no warnings, ...).
- Avoidance of language pitfalls.

Software coding standards - layout example

```
#ifdef SCHLECHTES_LAYOUT
struct TPaar {int iX;int iY;} tPaar;   /* Regel 1 verletzt */
int iWert1 ;                           /* Regel 1 verletzt */
int iWert2 = 1,iWert3 = 2;             /* Regel 1 verletzt */

tPaar. iX = 12;                        /* Regel 2 verletzt */
tPaar.iY =tPaar.iX + 19;               /* Regel 3 verletzt */
iWert1 = iWert2 ++ + ++ iWert3;        /* Regel 4 verletzt */
{
int* piWert2 = &iWert1,                /* Regel 5 verletzt */
     iScheinbarZeigerVariable;
}
assert(iWert1!=iWert2);                /* Regel 6 verletzt */
iWert3 = iWert1 * iWert2+3;            /* Regel 7 verletzt */
iWert3 = iWert1 *iWert2 + 3;           /* Regel 7 verletzt */
```


Software coding standards - layout example

```
#ifdef BESSERES_LAYOUT

int iWert1;
int iWert2 = 1, iWert3 = 2;
tPaar.iX = 12;
tPaar.iY = tPaar.iX + 19;
iWert1 = iWert2++ + ++iWert3;
{
   int *piWert2 = &iWert1,
       iKeineZeigerVariable;
}
assert(iWert1 != iWert2);
iWert3 = iWert1*iWert2 + 3;
iWert3 = iWert1 * iWert2 + 3;
```

Software coding standards - design example

```
int Horror(int);

main()
{
   int i = 12;
   int piArray[10];

   goto InDieSchleife;
   for (i = 0, Horror(1); i <= 10; i++)  /* Abbruchbedingung = pfui! */
   {
      printf("%d, ", i);
      piArray[i] = 0;   /* Schleifenzähler schlecht gewählt! */
InDieSchleife:
      printf("Schleife wird nie initialisiert.\n"
             "Horror() wird nicht aufgerufen\n" , i);
   }
}
```


Software coding standards - language pitfalls

```
inf fast_division(int i; unsigned j)
{
   return (i >> j);
}

/* the return value might be different
when using a different compiler */
```


Code review

Optional excercise:

Note the most important points you usually check in a code review apart from the ones tackled in the software coding standard section before.

Code review checklists

Make sure

- the software coding standard is followed.
- there are no double negations.
- debug code has been removed.
- the data types match the required accuracy.
- all variables are declareed within the correct scope.
- array indexes are not wrong by one and can not exceed the range or be negative.
- arithmetic overflows are handled.

Code review checklists

Make sure

- there are no implicit type casts.
- a division by 0 cannot happen.
- known compiler bugs have no effect.
- the code is free from language specific pitfalls and errors made in the past.
- ... (many more points)

→ Checklists improve the quality of code reviews.

Review of the IOPS firmware sources

Review items: main.c, debounc.h, debounc.c, dio.h, dio.c

Other files: system.h, messages.h

... intentionally **no** checklist!

Unit 5

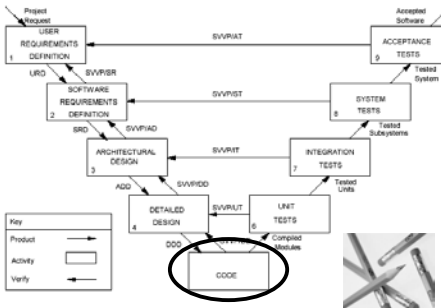
Verification of Low Level Software

Contents

Unit 5:

- *Code review meeting (software inspection)*
- Capabilities of static code checking tools
- *Application of a code checker to the reviewed software*
- *Follow-up review, metric calculation, and static check*
- Unit test design.
- Host target testing
- *Getting started with Cantata and the TCD file format*

IOPS code review meeting



Static code checker demo program (1/2)

```
#include <string.h>

char* MyName();                          /* 01 */
char cUartData = (char) 0xFA;            /* 02 */

main()
{
   unsigned uiA = (unsigned) cUartData;  /* 03 */
   int i, a[10] = {0,0,0,0,0,0,0,0,0,0};  /* 04 */

   for (i = 0; i <= 10; i++)             /* 05 */
      a[i] += i;                         /* 06 */

   for (i = 0; i <= 3; i++);             /* 07 */
   {
      char szString[10];                 /* 08 */
      a[i] -= cUartData;                 /* 09 */
      strcpy(szString, MyName());        /* 10 */
   }
}
```

Which code lines are obviously wrong?

Static code checker demo program (2/2)

```
char* MyName(int i)
{
   char name[11] = "Joe Jakeson";        /* 11 */
   if (i) name[2] = 'i';                 /* 12 */
   return name;                          /* 13 */
}
```

Which code lines are obviously wrong?

Solution to the static code checker demo

Capabilities of static code checkers

- Intendation checking
- Finding of common errors
- Initialization tracking
- Value tracking
- Evaluation order checking
- Function semantics
- Strong types
  - Index types
  - Hierarchies of types

```
if (x)
   if (y)
      statement();
else
   something_else();
```


Capabilities of static code checkers

- Intendation checking
- Finding of common errors
- Initialization tracking
- Value tracking
- Evaluation order checking
- Function semantics
- Strong types
  - Index types
  - Hierarchies of types

```
char cUartData = (char) 0xFA;

main()
{
   unsigned uiA = (unsigned) cUartData;
   ...
}
```


Capabilities of static code checkers

- Intendation checking
- Finding of common errors
- Initialization tracking
- Value tracking
- Evaluation order checking
- Function semantics
- Strong types
  - Index types
  - Hierarchies of types

```
void shame(void)
{
   int a,b,c,m,n,p;

   scanf("%d", &a);

   if (a) b = 6; else c = b;
   a = c;

   while (n--)
   {
      /* ... */
      m = 6;
      /* ... */
   }
   p = m;
}
```


Capabilities of static code checkers

- Intendation checking
- Finding of common errors
- Initialization tracking
- Value tracking
- Evaluation order checking
- Function semantics
- Strong types
  - Index types
  - Hierarchies of types

```
int f(int k, int n)
{
   if (k >= 10) a[0] = n;
   return a[k];
}

int *g(int *p)
{
   if (p) printf("\n");
   printf("%d", *p);
   return p + 2;
}
```


Capabilities of static code checkers

- Intendation checking
- Finding of common errors
- Initialization tracking
- Value tracking
- Evaluation order checking
- Function semantics
- Strong types
  - Index types
  - Hierarchies of types

```
a[i] = i++;

f(i++, n + i);
```


Capabilities of static code checkers

- Intendation checking
- Finding of common errors
- Initialization tracking
- Value tracking
- Evaluation order checking
- Function semantics
- Strong types
  - Index types
  - Hierarchies of types

```
{
   char buf[100];
   FILE *f;

   if (name) printf("ok\n");
   f = fopen(name, "r");
   fclose(f);
   /* char *fread(char *, size_t,
                  size_t, FILE *); */
   fread(buf, 100, 2, f);
}
```

Capabilities of static code checkers

- Intendation checking
- Finding of common errors
- Initialization tracking
- Value tracking
- Evaluation order checking
- Function semantics
- Strong types
  - Index types
  - Hierarchies of types

```
typedef unsigned short Att_Char;
typedef Att_Char Row[80];
typedef Row Screen[25];
typedef int Row_Ix;
typedef int Col_Ix;
#define BLANK (Att_Char) (0x700 + ' ')

Screen scr;
Row_Ix row;
Col_Ix col;
int i = 0;

scr[ row ][ col ] = BLANK;   // okay
scr[ i ][ col ] = BLANK;     // warning
scr[col][row] = BLANK;       // 2 wrngs
```


Capabilities of static code checkers

- Intendation checking
- Finding of common errors
- Initialization tracking
- Value tracking
- Evaluation order checking
- Function semantics
- Strong types
  - Index types
  - Hierarchies of types

```
typedef int voltage;
typedef int current;
typedef current leak_current;

current c;
voltage v;
leak_current lc;
int nonsense;

nonsense = c + v;  // no strong types

c = lc;  // works even in strong
         hierarchy

lc = c;  // works when wanted
```


Capabilities of static code checkers

Can static code checking programs replace a human review partner?

Can a human review partner make a static code checking program superfluous?

Application of a code checker to the reviewed software

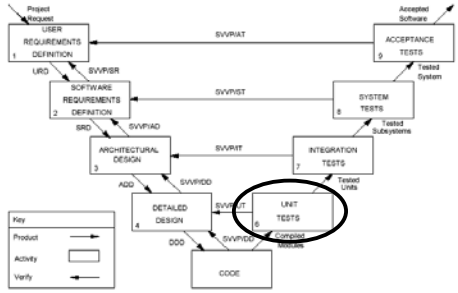
```
C:\LVA\Review> lint
```

Check if you find a bug you haven't found in the manual review!

Follow-up review of the IOPS Software

- Calculate code metrics and decide on readiness for code inspection.
- Re-run the static code checker.
- Re-do the code review. Inspect the fixes of the review item discrepancies.
- Assess readiness for unit testing.

Unit Testing



Unit test case design

- Retrieve the design specification of the unit under test. Derive black box test cases like for functional testing (see unit 2).
- Enhance the tests with **white box test cases** to satisfy a required coverage criteria.
- Test things that came up in the code review.

Statement Coverage

$$\text{Statement coverage} = \frac{\text{number of instrumented statements executed}}{\text{total number of instrumented statements}}$$

```
...
if (a && b) printf("hello");
...
```

Testcase 1: `a = 0, b = 0`.

→ no pracical meaning

Decision Coverage, Branch Coverage

$$\text{Decision coverage} = \frac{\text{number of decision outcomes exercised}}{\text{total number of decision outcomes}}$$

Decision coverage measures the proportion of "decision outcomes" which have been executed. A decision outcome is the evaluation of a control-flow condition to true or false, or jumping to a case or default label in a switch statement. Thus, each if, for, while or do..while construct corresponds to 2 decision outcomes (true and false), and each switch statement corresponds to N+1 decision outcomes (where there are N cases and a default case).

→ relation to McCabe's complexity metrics

Decision Coverage, Branch Coverage

$$\text{Decision coverage} = \frac{\text{number of decision outcomes exercised}}{\text{total number of decision outcomes}}$$

```
...
if (a && b) printf("hello");
...
```

Testcase 1: `a = 0, b = 1`.

Testcase 2: `a = 1, b = 1`.

Condition Coverage

$$\text{Condition coverage} = \frac{\text{number of condition outcomes exercised}}{\text{total number of condition outcomes}}$$

A condition is either a decision (except case statements) or is the operand of a Boolean operator, and is not itself the result of a Boolean operator. Each condition has two possible outcomes: *true* and *false*.

Condition Coverage

$$\text{Condition coverage} = \frac{\text{number of condition outcomes exercised}}{\text{total number of condition outcomes}}$$

```
...
if (a && b) printf("hello");
...
```

Testcase 1: `a = 1, b = 0`.

Testcase 2: `a = 0, b = 1`.

→ No practical meaning as stand-alone test metric!

Condition/Decision Coverage

Condition coverage = Min(decision coverage, condition coverage)

Condition/decision coverage combines the requirements for decision coverage with those for condition coverage. That is, there must be sufficient test cases to toggle the *decision* outcome between true and false and to toggle each *condition* value between true and false.

Condition/Decision Coverage

Condition coverage = Min(decision coverage, condition coverage)

```
...
if (a && b) printf("hello");
...
```

Testcase 1: `a = 0, b = 0`.

Testcase 2: `a = 1, b = 1`.

Modified Condition/Decision Coverage

The MC/DC criterion enhances the condition/decision coverage criterion by requiring that each condition be shown to independently affect the outcome of the decision. The independence requirement ensures that the effect of each condition is tested relative to the other conditions. However, achieving MC/DC requires in general, a minimum of n+1 test cases for a decision with n inputs. For the example (**A *or* B**), test cases (*TF*), (*FT*), and (*FF*) provide MC/DC. For decisions with a large number of inputs, MC/DC requires considerably more test cases than any of the coverage measures discussed before [NASA/TM-2001-210876].

Modified Condition/Decision Coverage

```
...
if (a && b) printf("hello");
...
```

Testcase 1: `a = 1, b = 1`.

Testcase 2: `a = 0, b = 1`.

Testcase 3: `a = 1, b = 0`.

Multiple Condition Coverage

Finally, multiple condition coverage requires test cases that ensure each possible combination of inputs to a decision is executed at least once; that is, multiple condition coverage requires exhaustive testing of the input combinations to a decision. In theory, multiple condition coverage is the most desirable structural coverage measure; but, it is impractical for many cases. For a decision with n inputs, multiple condition coverage requires 2^n tests.

Multiple Condition Coverage

```
...
if (a && b) printf("hello");
...
```

Testcase 1: `a = 1, b = 1`.

Testcase 2: `a = 0, b = 1`.

Testcase 3: `a = 1, b = 0`.

Testcase 4: `a = 0, b = 0`.

Unit Testing on the test

Given you test a unit and have reached multiple condition coverage. Is this a sufficient criteria to stop unit testing?

Unit Testing on the test

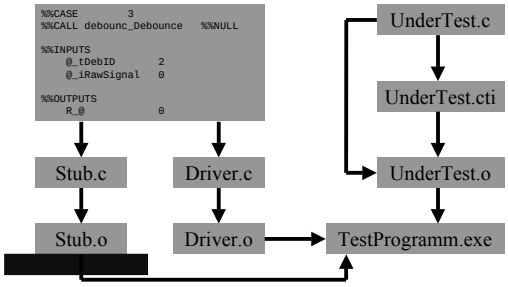
How would you verify the following code?

```
unsigned short intsqrt(unsigned uiInput)
{
   unsigned short hiRoot = 0;
   unsigned short uiRemHi = 0, uiRemLo = uiInput;
   unsigned short uiTestDiv;
   short iBits;

   for(iBits = 0; iBits < 16; iBits++)
   {
      uiRemHi = (uiRemHi << 2) | (uiRemLo << 30);
      uiRemLo <<= 2;
      hiRoot <<= 1;
      uiTestDiv = (hiRoot << 1) + 1;
      if (uiRemHi >= uiTestDiv)
      {
         uiRemHi -= uiTestDiv;
         hiRoot++;
      }
   }
   return hiRoot;
}
```

Host target testing

Recall from unit 3 (isolation testing):



Host target testing

Why to run unit tests in the host environment?

- A bottleneck may form of many developers competing for time on the target environment in order to test software. The target environment may not yet be available.
- Target environments are usually less sophisticated and less convenient to work with than host environments.
- Instrumented code and test program might be too big for the target environment.
- No `printf` of test results ...

Host target testing

Why should one run unit tests on the target?

- When unit tests are re-run for integration testing, there is no other option (see unit 3).
- The use of inline Assembler leaves no other choice.
- Calls to the target operating system do not need to be stubbed.
- Real world input data can be used.
- C-library bugs and compiler bugs can be revealed.
- The length of the `int` data type is correct.

Host target testing

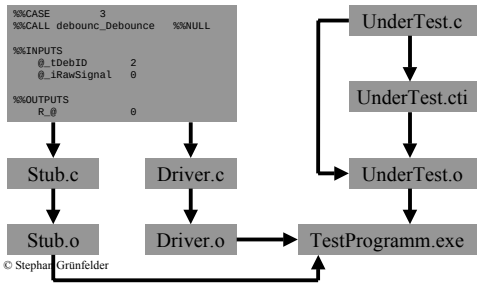
A suggested strategy for crosstesting is:

(1) Execute tests in debug mode, using instrumented software. If possible in the host environment. Ensure that coverage objectives are achieved. Correct any errors in the software and in the test script.

(2) Verify that tests execute correctly in the target environment by repeating tests in target environment, using uninstrumented software compiled with the final compiler switches.

Getting started with Cantata

Total recall ... ☺



Getting started with Cantata

Testing the file \LVA\maths4\maths.c, see maths.tcd:

```
//////////////////////////////////////////
%%CASE                  1
%%COMMENT               do a square of an in-range value

%%CALL                  maths   range#1

%%INPUTS
maths_op                SQUARE
maths_value             100
call_count              5

%%OUTPUTS
M_maths_result          10000
R_maths                 NO_ERROR
call_count              6
//////////////////////////////////////////
```


Getting started with Cantata

Abbreviations

```
//////////////////////////////////////////
%%CASE                  1
%%COMMENT               do a square of an in-range value

%%CALL                  maths   range#1

%%INPUTS
@_op                    SQUARE
@_value                 100
call_count              5

%%OUTPUTS
M_maths_result          10000
R_@                     NO_ERROR
call_count              6
//////////////////////////////////////////
```


Getting started with Cantata

Testing the file \LVA\maths4\maths.c, see maths.tcd:

```
//////////////////////////////////////////
// Stubs Section
//////////////////////////////////////////

%%STUB                  range

%%ACTION                1
%%INPUTS
@_value                 100
%%OUTPUTS
R_@                     RANGE_OK

//////////////////////////////////////////
```


Getting started with Cantata

- To set up the Cantata environment, run begin.bat in a Command Window.
- To make a test for source file src.c with test case definition file test.tcd, type
  mktest src test
  in the command window.
- To execute the test, start the created executable.

Getting started with Cantata

Complete maths.tcd to reach 100% decision coverage!

```
================================================================================
Tests Completed At : 08:25:33
--------------------------------------------------------------------------------
Test   Script   Checks   Checks   Checks   Stubs   Paths   Assertions   Status
       Errors   Passed   Failed   Warning  Failed  Failed  Failed
--------------------------------------------------------------------------------
PTE       0        0        0        0        0       0       0          PASS
001       0        4        0        0        0       0       0          PASS
002       0        3        0        0        0       0       0          PASS
003       0        4        0        0        0       0       0          PASS
ANS       0        2        0        0        0       0       0          PASS
--------------------------------------------------------------------------------
Total     0       19        0        0        0       0       0          PASS
================================================================================
```

Getting started with Cantata

- Inspect the intrumented source file maths.cti
- Inspect the test code mathst.c
- Why does the source code use `STATIC` instead of `static`?

© Stephan Grünfelder

Unit 6

Verification of Low Level Software

Contents

Unit 6:

- *Implementation of simplified unit tests* (moderated), *i.e. finish the maths.tcd example from unit 5.*
- A note on test metrics.
- Tools for automatic test case generation.
- *Design and implementation of unit tests for the IOPS project*

Complete maths.tcd ...

```
================================================================================
Tests Completed At : 08:25:33
--------------------------------------------------------------------------------
Test   Script   Checks   Checks   Checks   Stubs   Paths   Assertions   Status
       Errors   Passed   Failed   Warning  Failed  Failed  Failed
--------------------------------------------------------------------------------
PTE       0        0        0        0        0       0       0          PASS
001       0        4        0        0        0       0       0          PASS
002       0        3        0        0        0       0       0          PASS
003       0        4        0        0        0       0       0          PASS
ANS       0        2        0        0        0       0       0          PASS
--------------------------------------------------------------------------------
Total     0       19        0        0        0       0       0          PASS
================================================================================
```


Solution for maths.tcd

..\maths4_solution\maths5_tcd.pdf

Testing test metrics

Invent a very simple function and test cases that achieve 100% Multiple Condition Coverage but nevertheless do not find the bug!

Testing test metrics – example solution

```
do_nothing(int a, int b)
{
   if (a) nop(); else x++;
   if (b) nop(); else x--;
}
```

Testcase 1: `a = 0, b = 0;`
Testcase 2: `a = 1, b = 1;`

Would structured testing find the problem?

Would structured testing be able to find all possible problems in this function?

Testing test metrics

Assume you can test all possible paths through a program. Can you have full confidence in the tested program by having all paths tested?

How long would it take to test all paths of a normal program?

Advanced test tools

Cantata is a tool that serves the teaching purpose but is superseded by (an) advanced product(s).

You can inspect what happens inside. Have a look at the CTI files, the mktest.bat and the test code generated!

What can state-of-the-art tools do better?

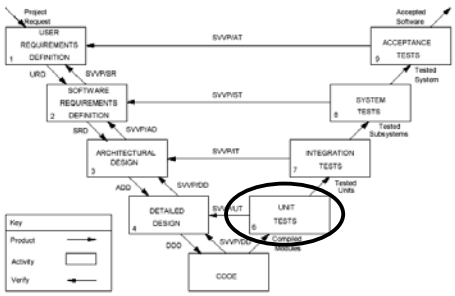
Advanced test tools

- Test tool is part of the IDE. No TCD files. The tool parses the source code and generates the TC environment by itself, i.e. empty stubs and drivers.
- No definition of expected output necessary. The tool executes the code with the defined input and suggests the expected output.
- In case a test case reveals an error, the debugger of the IDE automatically jumps into the affected function.

Advanced test tools

- The test tool is fully integrated with the In Circuit Emulator. If more test cases are defined, the size of the executable will not increase.
- Graphical display of boundary value analysis as a classification tree.
- Very fancy tools can even automatically generate test cases on numerical boundaries.

IOPS unit testing



IOPS unit testing

Test the source file debounc.c in an isolation testing approach. 100% decision coverage is required. Note that we have an OO design in plain C-language!

IOPS unit testing – the bugs

© Stephan Grünfelder

Unit 7

Verification of Low Level Software

Contents

Unit 7:

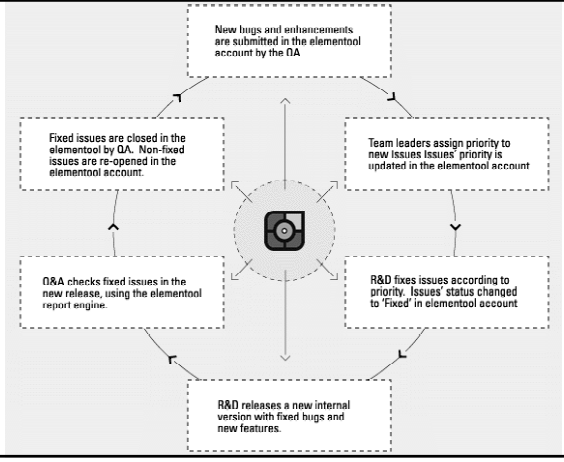
- *Design and implementation of IOPS unit tests*
- Bug tracking
- Generating and using unit test error statistics
- A note on OO Test metrics
- Unit test summary report
- *System test readiness review*

IOPS unit testing (continued)

Test the source file debounc.c in an isolation testing approach. 100% decision coverage is required. Note that we have an OO design in plain C-language!

IOPS unit testing – the bugs

Bug tracking lifecycle from a web ad


New bugs and enhancements are submitted in the elementool account by the DA
Team leaders assign priority to new issues Issues' priority is updated in the elementool account
R&D fixes issues according to priority. Issues' status changed to 'Fixed' in elementool account
R&D releases a new internal version with fixed bugs and new features.
Q&A checks fixed issues in the new release, using the elementool report engine.
Fixed issues are closed in the elementool by QA. Non-fixed issues are re-opened in the elementool account.

Bug tracking tools

- Reporting and assignment of priorities.
- Routing of bugs.
- Email notification.
- WWW-based tools.
- What defects are currently open?
- What are the defects I am in charge of?
- What entries did not change for more than a month?
- What defects have been fixed?

A collection of tools can be found at http://www.sqatester.com/bugtracking/

Generating and using ((unit test) error) statistics

Test assessment steps [Perry95]

- Establish assessment objectives
- Identify what to measure
- Assign measurement responsibilities
- Select approach to evaluation
- Identify needed facts
- Collect evaluation data
- Assess the effectiveness of testing

Example test metrics

- Test coverage (see unit 5)
- Test progress (instructions exercised/total number of instructions)
- Test costs
- Cost to locate defect
- Error ratio (number of errors detected versus software system/unit size)
- Test efficiecy (defects located by testing versus total system defects)
- Effectiveness of testing (loss due to problems versus test costs)
- Rerun analysis (rerun hours versus production hours)
- Test automation ratio

OO test metrics

Note that conventional coverage analysis does not show the full picture in OO programming.

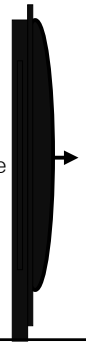
Does testing `base.foo()` mean that `derived.foo()` works?

→ inheritance context decision coverage, see [IPL99]

(Unit) test summary report

Test results may be reported in a variety of ways. Some common means of recording results are:

- test result forms, recording the tester, date and outcome of the test cases executed by the procedure;
- execution logfiles.



Who wants to read all that?

Managers and SQAs want summary reports.

Contents of the test summary report

- Version of software under test, test version.
- Tester,date.
- Used test environment (especially when testing an embedded system).
- Summary of the test results itself, e.g.

  ST-FanCo-Ft04 fails because at the moment there are no duty cycles sent via CAN. This is planned for phase 3.
  ST-Alt-FT01 fails: when stalling the engine by mistake a MIN error occurs at DFP_AltExc1.
  All other tests pass without problem.

Unit 8

Verification of Low Level Software

Contents

Unit 8:

- Generating and using system test error statistics
- *Running the designed IOPS system tests*
- System Test Summary Report
- Extreme Programming
- Risk based testing
- Conclusion
- *Feedback*

Generating and using system test error statistics

Why statistics on system testing?

- Find weaknesses in previous testing steps.
- Find weaknesses of the system test process (if flaw was found after testing).
- Assess readiness for delivery.
- Questions of cost and effectiveness as presented in unit 7.

Generating and using system test error statistics

Why statistics on system testing?

- Find weaknesses in previous testing steps.
- Find weaknesses of the system test process (if flaw was found after testing).
- Assess readiness for delivery.

→ Test process should be questioned.

→ Personnel must be trained.

→ More caution must be taken with certain software modules.

Generating and using system test error statistics

Why statistics on system testing?

- Find weaknesses in previous testing steps.
- Find weaknesses of the system test process (if flaw was found after testing).
- Assess readiness for delivery.

→ Test process must be improved.

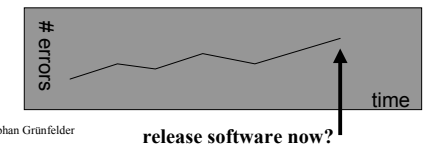
→ Personnel must be trained.

→ Unabigousness of specifications should be checked.

Generating and using system test error statistics

Why statistics on system testing?

- Find weaknesses in previous testing steps.
- Find weaknesses of the system test process (if flaw was found after testing).
- Assess readiness for delivery.

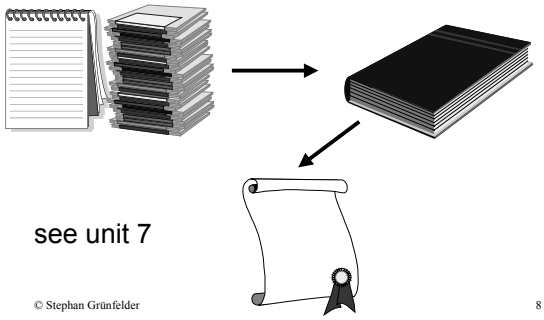


Running the designed IOPS system tests

Use either your personal systemt tests or the sample system test document and break the IOPS software!

PP-bus simulator commands:
o ... open oil pressure switch
c ... close oil pressure switch
b ... send next command from rx_bus.txt

System Test Summary Report



see unit 7

Extreme Programming (XP)

Extreme Programming [Beck01]

- introduces pair programming instead of code reviews;
- implements tests before coding;
- instead of writing SRDs the requirements are captured in test scenarios;
- has many more iterations than other lifecycle models.

XP, the SQA, and embedded systems

- Pair programming: harder to identify common errors and difficult to *prove* code review.
- Implementing tests before coding might add costs when software design is infeasible because of hardware problems.
- Many iterations and early software deliveries might be impossible in in projects with hardware/software co-design.

Risk based testing

Idea:

- Assess likelihood of a flaw in program fragments, the impact, and visibility of the flaw.
- Distribute test effort dependent on a risk estimation based on the above factors.

Risk likelihood assessment

Scale	Size/Complexity	Number of Former Changes	Level of Optimization (Resources)	Defects Found Earlier
1	very small	established functionality, never changed	no optimization	never
3	small	infrequently revised	less optimization	infrequently
6	large	frequently revised	heavy optimization	frequently
9	very large	permanently revised	very heavy optimization	permanently

How could this be more precise?

Other factors: experience of developers, change frequency of requirements, ...

Impact assessment

Scale	Safety
1	no component protection and not MFT relevant
6	component protection
9	MFT relevant

Scale	Customer Importance
1	functionality is nice to have
3	functionality is standard requirement
6	functionality is important (e.g. new functionality based on changed laws)
9	functionality is very important for the customer (e.g. a functionality which differs the customer from other competitors)

Visibility assessment

Risk based testing is the application of Risk Management to testing. [Hall03] defines a surprise factor, as the likelihood that the risk is visible.

$\rightarrow$ How likely do we detect a bug in an operational system?

Risk based testing

A possible score for the testing effort requirement is

likelihood $\times$ impact $\times$ surprise

Similar scores may be used to select the tool, test coverage, test team, automation extent, ...

(see example guideline appended to lecture notes)

Conclusion

- Verification, Testing, Validation are related techniques but different things.
- Testing is a creative job. Experience and training is needed.
- Testing needs to be planned.

Conclusion

- Different bugs are found in different phases of the software development with different techniques (document reviews, code inspection, UT, IT, ST).
- No single of the above mentioned techniques can claim to find all bugs.
- **IT IS ALL ABOUT MONEY**. Tune your testing according to the € you get.

Feed back session

What did you like?

What did you already know before?

What did you miss?

What was too easy?

What was too hard to understand?

What was too boring?

Would you recommend the course?

Any general suggestions?

Contact

Stephan Grünfelder
Rosentalgasse 3/1/3
A-1140 Wien

stephan.gruenfelder@gmx.at

© Stephan Grünfelder

TheCompany Ltd.
Rosentalgasse 3
A-1140 Wien

PLS – Intelligent Oil Pressure Sensor

Software Requirements Document

Document: **004589**
Issue: **1.0**
Date: **28 Aug 2003**

	Name:	Function:	Signature:	Date:
Prepared:	Stephan Grünfelder	SW Designer	……	……..
Checked:	Bernd Spiess	Project Engineer	……	……..
	Robert Stolz	Verification Engineer	……	……..
	Helga Machart	SW Quality Assurance	……	……..
Approved:	Norbert Wichtig	Project Manager	……	……..

Document Distribution List

Hansi Hinterseer, Arnold Schwarzenegger, Cindy Crawford, Hubert von Goisern

Change Log

Version, Date	Change Description
1.0, 28 Aug 2003	new document

Table of Contents

1 Intoduction 4
1.1 Purpose 4
1.2 About this Issue 4
1.3 Acronyms 4
2 Documents 4
2.1 Applicable Documents 4
2.2 Reference Documents 5
3 Definitions 5
3.1 Requirement Format and Numbering 5
3.2 Typographic Conventions 5
3.3 General Technical Terms and Phrases 6
4 General Description 6
4.1 IOPS Environment 6
4.2 Purpose and General Functional Principle 6
4.3 Hardware 7
5 Specific Software Requirements 7
5.1 Functional Requirements 7
5.1.1 Requirements on the Initialisation Process 7
5.1.2 Requirements on the PP-bus Communication 7
5.1.3 Requirements on the Sensor Interface 8
5.1.4 Reliability and Accuracy 9
5.2 Software Budget 9
5.3 Software Design 9
6 Traceability 10

1 Intoduction

1.1 Purpose

This document is part of the Project Little Stream (PLS), a sensor network for medium sized water power plants. As a Software Requirements Document (SRD) it is the baseline for the software development for the Intelligent Oil Pressure Sensor (IOPS), which reports the oil pressure state of a turbine on the PowerPlant-bus.

1.2 About this Issue

Issue 1.0 of the document was prepared for the PLS IOPS System Requirements Review. It is mainly based on the PLS Marketing Requirements Document [MRD] and on the PowerPlant-bus Interface Control Document [ICD].

1.3 Acronyms

AD	Applicable Document
CPU	Central Processing Unit
IOPS	Intelligent Oil Pressure Sensor
LSB	Least Significant Bit
PCB	Printed Circuit Board
PLS	Project Little Stream
PP-bus	PowerPlant bus, a company standard to interconnect sensors in a power plant
RD	Reference Document
SRD	Software Requirements Document

2 Documents

In the event of a conflict between this document and an Applicable Document (AD), the AD shall have the precedence. In the event of a conflict between this document and a Reference Document (RD), this document shall have precedence.

Any such conflict should, however, be brought to the attention of TheCompany Ltd. for resolution.

This document has been established based on the issues of ADs and RDs as given below. Issue changes of ADs and RDs will lead to an update of this document only in case of impacts on its content.

The valid documentation status is reflected in the relevant Configuration Items Data List.

2.1 Applicable Documents

[MRD]	TheCompany, 004537	Little Stream Marketing Requirements Document V2.1
[ICD]	ABB, IA-PL-1520-00	PowerPlan-Bus Interface Control Document, Release 1.0
[HW]	TheCompany, 004545	IOPS PCB Design 1.0

2.2 Reference Documents

[CS1] F. Griesauer: Guter Code braucht Ordnung, Teil 1: Universelle Regeln zur Namensgebung und zu Zahlenformaten. *Elektronik*, Heft 18/2001, S. 52-59.

[CS2] S. Programmierer: Guter Code braucht Ordnung, Teil 2: Das Code-Layout. *Elektronik*, Heft 8/2002, S. 74-81.

[CS3] A. Schlau: Guter Code braucht Ordnung, Teil 3: Fehlervermeidung bei der Erstellung von C-Programmen. *Elektronik*, Heft 14/2002, S. 66-71.

[DSTD] The Company, 003720 Design Standards for Embedded Systems Software 2.0

3 Definitions

3.1 Requirement Format and Numbering

The requirements in this document are stated between a requirement number and a requirement delimiter.

The requirement number follows the subsequent syntax rule

R/SRD/<section>/<number>-<issue>

where

<section> is an identifier for the chapter, section, or subsection to which the requirement belongs to

<number> is an incremental number used within each instance of <section>

<issue> is a number indicating in which issue of the specification the requirement was last changed. Initially it is set to '1'.

The requirement delimiter is a parallelogram (#).

If a requirement is no longer valid it is replaced by the text "--- deleted". Requirement numbers may not be removed or reused.

The verb *shall* is used whenever the described practice is compulsory, *should* is used for recommended practices. Sentences without any of the two verbs within a formal requirement are explanations.

3.2 Typographic Conventions

Text in *italic face* represents a term that is defined or has has special meaning within this document. An instance of such a term is not necessarily in italic face, if the context will leave no doubt that the definition is applicable.

Bold face letters emphasise the textual content and improve readability.

Binary numbers are tagged with a subscript 'b', like 0101_b.
Hexadecimal numbers are tagged with a subscript 'h', like $3FA9_h$.
Numbers without subscripts are to be interpreted as decimals.

3.3 General Technical Terms and Phrases

In the following table a definition of used technical terms and phrases is found.

Term/Phrase	Meaning
arm a watchdog	reset a watchdog timer and start its clock
clear a bit	make a bit zero
disarm a watchdog	stop the clock of a watchdog timer
kick a watchdog	reset the watchdog timer, i.e. reload it with a previously defined value
mask an interrupt	disable interrupt serving but latch the interrupt
PP-bus	a company standard interconnection for sensors in a power plant
set a bit	make a bit one

Table 1: Technical terms used in this document.

4 General Description

4.1 IOPS Environment

The objective of PLS is to provide the power plant designer with a set of sensors that all can communicate with the bus master, the main control unit of the power plant. The Intelligent Oil Pressure Sensor is one of them. The IOPS hosts a mechanical switch that opens if the oil pressure of the turbine lubrication is lower than a temperature dependent threshold, and closes if it is higher. In normal operation the switch should be closed and an open switch would indicate a failure in the lubrication system. However, when the turbine runs on low speed it might open temporarily, and when the turbine stops it must open – otherwise it would mean that the switch is stuck.

The main control unit gets the status of all sensors at regular intervals via status messages on the PP-bus.

4.2 Purpose and General Functional Principle

After power-up the IOPS gets programmed. The debounce times and report rates are negotiated via the PP-bus. Upon a start command from the main control, the MC4711 Micro Controller starts transmitting the debounced state of the oil pressure switch at the negotiated rate (in the range of seconds). Primarily this messages are sent to show that the IOPS is alive. However, whenever the debounced oil pressure switch state changes, the IOPS immediately sends a state message to the main control (which must start appropriate actions).

During start-up the MC4711 performs a self-test. If it fails, then the IOPS does not respond to any command to show it is broken. In case of errors during the operation the MC4711 reboots.

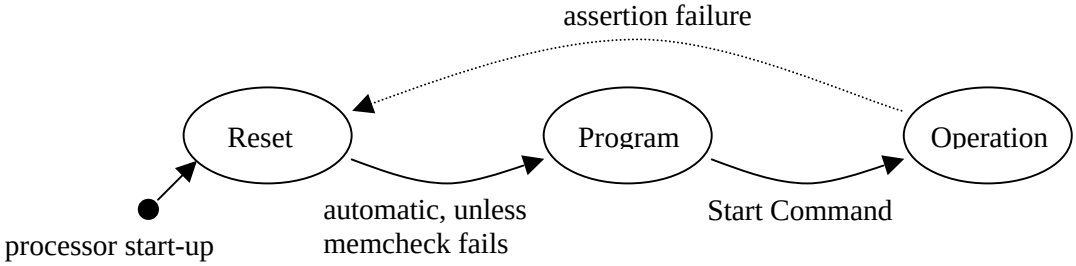


Figure 1: State transitions of the IOPS.

4.3 Hardware

The IOPS is based on the MC4711, which implements a fully operational PP-bus controller. Thus, issues like bus arbitration, address decoding, and message checksums are fully transparent to the firmware. The MC4711 is a 16-bit controller running at 10MHz with 16kB on-board RAM and 16kB EEPROM. It also contains a digital I/O port. One of its pins is connected to the mechanical switch, one to an external watchdog, and one to a light emitting diode [HW].

5 Specific Software Requirements

This chapter contains the formal requirements on the IOPS software.

5.1 Functional Requirements

5.1.1 Requirements on the Initialisation Process

R/SRD/INIT/10-1: Self Test
After reset a selft-test is performed that checks the RAM and the processor core. #

R/SRD/INIT/20-1: Self Test Fail Silence
The IOPS software shall only reply to PP-bus commands when the self-test at start-up passes. #

5.1.2 Requirements on the PP-bus Communication

Each PP-bus message consists of 4 bytes. The first byte is an address byte, it is followed by two *data bytes* and a checksum byte. The first and last bytes are stripped off by the PP-bus controller.

R/SRD/PP/10-1: Message Definitions
The first data bytes shall be a message identifier according to the following table and the second data byte shall represent a parameter byte. #

Die Beschreibung “the fist bytes shall be a msg id” ist eine Unsinn. Darauf hat die IOPS SW keinen Einfluss. Dieses Req. ist völlig redundant zu anderen.

ID	meaning	direction
01_h	start being operational	control unit to sensor
02_h	set positive debounce time	control unit to sensor
03_h	set negative debounce time	control unit to sensor
04_h	set the state report interval	control unit to sensor
20_h	oil switch state report	sensor to control unit
23_h	acknowledge programming parameter	sensor to control unit

Table 2: IOPS related PP-bus messages and meanings.

R/SRD/PP/20-1: PP-bus Performance
Upon receipt of a command via the PP-bus, the IOPS shall reply within 1ms (measured between the first byte sent and the last byte received). #

R/SRD/PP/30-1: Reply Message Format
The first *data bytes* of the reply message of the IOPS shall be the ID 23_h. The second byte shall be a copy of the first *data byte* of the command the IOPS is replying to. #

R/SRD/PP/40-1: SetPosDebTime Command
Upon receipt of the data bytes

1st byte	2nd byte
02_h	POS

the IOPS software shall set the positive debounce time for the oil pressure switch to an appropriate offset + POS, where POS shall be interpreted as a signed two-bit complement number and the programmable range shall be from 0 to 2 seconds. #

Comment: the positive debounce time is the debounce time for the transition closed to opened switch.

R/SRD/PP/50-1: SetNegDebTime Command
Upon receipt of the data bytes

1st byte	2nd byte
03_h	NEG

the IOPS software shall set the negative debounce time for the oil pressure switch to an appropriate offset + NEG, where NEG shall be interpreted as a signed two-bit complement number and the programmable range shall be from 0 to 2 seconds. #

Comment: the negative debounce time is the debounce time for the transition opened to closed switch.

R/SRD/PP/60-1: SetRepIntv Command
Upon receipt of the data bytes

1st byte	2nd byte
04_h	SEC

the IOPS software shall report the *debounced oil pressure switch state* at intervals of min(SEC,30) seconds as soon as it has been switched to operational mode by the message of R/SRD/PP/70. #

Eigentlich bräuchte man nur eine message, die beides glchztg. macht: start und SetRepIntv

R/SRD/PP/70-1: Start Command
Upon receipt of the data bytes

1st byte	2nd byte
01_h	XX

the IOPS software shall start reporting the *debounced oil pressure switch state* at intervals specified in R/SRD/PP/60. The IOPS software shall not care about the 2^{nd} byte. #

Shall not care about the 2^{nd} byte ist ein Nicht-Requirement und so nicht testbar

R/SRD/PP/80-1: Report Message Format
The IOPS shall report the *debounced oil pressure switch state* by sending the following PP-bus message

1st byte	2nd byte
20_h	STC

where STC = 1 in case of an open switch and STC = 0 in case of a closed switch.

5.1.3 Requirements on the Sensor Interface

Low oil pressure opens the oil pressure switch and the LSB of the digital I/O port register of the MC4711 becomes 1. High oil pressure closes the switch, the LSB of the I/O port register becomes zero [HW]. To ease the distinction in the following the term *position* will be used for the physical state of the sensor (the raw sensor value), whereas the debounced switch *state* is what the IOPS reports.

R/SRD/SI/10-1: Switch Position Sampling Rate
The IOPS software shall sample the position of the oil pressure switch at a rate of 1kHz. #

R/SRD/SI/20-1: Switch Position Spikes
To compute the *debounced oil pressure switch state* the IOPS software shall use an algorithm that is tolerant to single outliers of the raw sensor position. #

Völlig redundant zu nächstem Requirement

R/SRD/SI/30-1: Debouncing Algorithm
To compute the *debounced oil pressure switch state* the IOPS software shall use an algorithm that integrates over the raw sensor values. If all raw values sampled within the debounce time correspond to a different state than the current debounced oil pressure switch state, then the state must flip and it must take at least the same number of samples of opposite switch positions to flip the state again. #

Nicht wirklich gut erklärt ein Beispiel, oder eine Grafik wäre nett. Das “must” ist auch ein bisserl verwirrend. Bei Revision besser weglassen.

5.1.4 Reliability and Accuracy

R/SRD/RA/10-1: Missing Parameters
In case the IOPS is not programmed within 10 seconds after start-up, a re-boot of the processor is necessary. #

Was heißt “programmed”? Das Start-Command oder irgendein Kommando?

R/SRD/RA/20-1: PP-bus Error
In case the software cannot interpret a PP-bus command or encounters an invalid software state, it shall initiate a reboot of the processor within one second. #

Comment: The software should be operable as much as possible, there is no need to log the cause of the error or stay fail silent, once the software is fully operational.

Dieser Kommentar ist eine Hilfe, damit man leichter erkennt was für ein Blödsinn hier gefordert ist: Wenn die IOPS SW stirbt und der WD rebootet, dann wartet der IOPS auf eine neue Parametrisierung anstatt entweder zu melden, dass er neu parametrisiert werden muss oder die alten Parameterwerte neu zu verwenden.

R/SRD/RA/30-1: Watchdog Kicking
The software shall kick a watchdog at least once a second and at most 10000 times a second by setting bit 1 of the digital I/O port for at least 50 clock cycles and by clearing the bit afterwards. #

Comment: the external watchdog will reset the MC4711 if it is not kicked for 1000ms. A deadlock resolved by the watchdog should in the best case not visible to other devices on the PP-bus.

Das wird ja wohl nicht erfüllt. Spätestens ab der FMEA, FTA wird das klar: bei einem WD reset in operational hört der IOPS auf seinen Zustand zu melden. Eigentlich ist das ein Requirement und kein Kommentar!

5.2 Software Budget

R/SRD/SB/10-1: Maximum CPU Load
The timing budget of the IOPS software shall load the CPU up to at most 80%. #

R/SRD/SB/20-1: Idle Time
If the MC4711 is not busy with processing, it shall execute the *idle command* to save power and prolong the lifetime of the micro controller. #

Comment: The *idle command* puts the MC4711 in a low-power state. The micro controller proceeds program execution after executing the idle command if and only if an unmasked interrupt has been served since the idle command.

R/SRD/SB/30-1: Memory Footprint
The IOPS software shall consume at most 80% of the MC4711 on-chip memory. #

5.3 Software Design

R/SRD/SD/10-1: Software Codig Standard
The IOPS software shall conform to the software coding standard of [CS1, CS2, CS3]. #

R/SRD/SD/20-1: Software Re-use
The IOPS software shall re-use as much source code from the KJ project as possible. #

R/SRD/SD/30-1: Programming Language
The IOPS software shall be programmed in C whenever possible. #

6 Traceability

This chapter captures the origin of the software requirements in tabular form.

Req#	Req Short Text	Origin
R/SRD/INIT/10	Self Test	[MRD] section 2.1
R/SRD/INIT/20	Self Test Fail Silence	[MRD] section 2.1
R/SRD/PP/10	Message Definitions	[ICD] R/ICD/MSG/010
R/SRD/PP/20	PP-bus Performance	[ICD] R/ICD/P/030
R/SRD/PP/30	Reply Message Format	[ICD] R/ICD/MSG/110
R/SRD/PP/40	SetPosDebTime Cmd	[ICD] R/ICD/MSG/130
R/SRD/PP/50	SetNegDebTime Cmd	[ICD] R/ICD/MSG/140
R/SRD/PP/60	SetRepIntv Cmd	[ICD] R/ICD/MSG/160
R/SRD/PP/70	Start Command	[ICD] R/ICD/MSG/105
R/SRD/PP/80	Report Message Format	[ICD] R/ICD/MSG/125
R/SRD/SI/10	Switch Position Sampling Rate	minutes of the technical meeting ABB/Thecompany, 12 Apr 2003, Göteborg
R/SRD/SI/20	Switch Position Spikes	minutes of the technical meeting ABB/Thecompany, 12 Apr 2003, Göteborg
R/SRD/SI/30	Debounciung Algorithm	e-mail 0815 from j.r.ewing@dallas.com to arnold@muskelkater.at
R/SRD/RA/10	Missing Parameters	[MRD] section 4
R/SRD/RA/20	PP-bus Error	[MRD] section 4
R/SRD/RA/30	Watchdog Kicking	[HW]
R/SRD/SB/10	Maximum CPU Load	[DSTD]
R/SRD/SB/10	CPU Idle Time	[MRD] section 2
R/SRD/SB/30	Memory Footprint	[DSTD]
R/SRD/SD/10	Software Codig Standard	[DSTD]
R/SRD/SD/20	Software Reuse	decisition of the project management, see TheCompany Mom 0009240, 24 Mar 2003
R/SRD/SD/30-1	Programming Language	for the MC4711 there is no other compiler available

TheCompany Ltd.
Rosentalgasse 3
A-1140 Wien

PLS – Intelligent Oil Pressure Sensor

Software Requirements Document

Document: **004589**
Issue: **2.0**
Date: **12 Sep 2003**

	Name:	Function:	Signature:	Date:
Prepared:	Stephan Grünfelder	SW Designer	……	……..
Checked:	Bernd Spiess	Project Engineer	……	……..
	Robert Stolz	Verification Engineer	……	……..
	Helga Machart	SW Quality Assurance	……	……..
Approved:	Norbert Wichtig	Project Manager	……	……..

Document Distribution List

Hansi Hinterseer, Arnold Schwarzenegger, Cindy Crawford, Hubert von Goisern

Change Log

Version, Date	Change Description
1.0, 28 Aug 2003	new document
2.0, 12 Sep 2003	changes acc. to the PLS IOPS System Requirements Review
	inserted new requirement R/SRD/PP/05 according to [MoM] RID 1
	deleted R/SRD/PP/10, acc to [MoM] RID 2
	R/SRD/PP/70: reworded the requirement
	deleted R/SRD/SI/20, acc to [MoM] RID 4
	refined R/SRD/SI/30 with a figure that explains the debouncing in detail acc. to [MoM] RID 5
	Refined R/SRD/RA/10.

Das Datum und die Ausgabenummer in der Kopfzeile stimmen nicht

Table of Contents

1 Intoduction 4
1.1 Purpose 4
1.2 About this and Previous Issues 4
1.3 Acronyms 4
2 Documents 4
2.1 Applicable Documents 4
2.2 Reference Documents 5
3 Definitions 5
3.1 Requirement Format and Numbering 5
3.2 Typographic Conventions 5
3.3 General Technical Terms and Phrases 6
4 General Description 6
4.1 IOPS Environment 6
4.2 Purpose and General Functional Principle 6
4.3 Hardware 7
5 Specific Software Requirements 7
5.1 Functional Requirements 7
5.1.1 Requirements on the Initialisation Process 7
5.1.2 Requirements on the PP-bus Communication 7
5.1.3 Requirements on the Sensor Interface 8
5.1.4 Reliability and Accuracy 9
5.2 Software Budget 9
5.3 Software Design 10
6 Traceability 10

1 Intoduction

1.1 Purpose

This document is part of the Project Little Stream (PLS), a sensor network for medium sized water power plants. As a Software Requirements Document (SRD) it is the baseline for the software development for the Intelligent Oil Pressure Sensor (IOPS), which reports the oil pressure state of a turbine on the PowerPlant-bus.

1.2 About this and Previous Issues

Issue 1.0 of the document was prepared for the PLS IOPS System Requirements Review. It is mainly based on the PLS Marketing Requirements Document [MRD] and on the PowerPlant-bus Interface Control Document [ICD].

Issue 2.0 is an update based on the review results [MoM].

1.3 Acronyms

AD	Applicable Document
CPU	Central Processing Unit
IOPS	Intelligent Oil Pressure Sensor
LSB	Least Significant Bit
PCB	Printed Circuit Board
PLS	Project Little Stream
RD	Reference Document
RID	Review Item Discrepancy
SRD	Software Requirements Document

2 Documents

In the event of a conflict between this document and an Applicable Document (AD), the AD shall have the precedence. In the event of a conflict between this document and a Reference Document (RD), this document shall have precedence.

Any such conflict should, however, be brought to the attention of TheCompany Ltd. for resolution.

This document has been established based on the issues of ADs and RDs as given below. Issue changes of ADs and RDs will lead to an update of this document only in case of impacts on its content.

The valid documentation status is reflected in the relevant Configuration Items Data List.

2.1 Applicable Documents

[MRD]	TheCompany, 004537	Little Stream Marketing Requirements Document V2.1
[ICD]	ABB, IA-PL-1520-00	PowerPlan-Bus Interface Control Document, Release 1.0
[HW]	TheCompany, 004545	IOPS PCB Design 1.0

[MoM] TheCompany, 004596 Minutes of Meeting of the external PLS IOPS System Requirements Review, Wr. Neutstadt, 8 Sep 2003

2.2 Reference Documents

[CS1] F. Griesauer: Guter Code braucht Ordnung, Teil 1: Universelle Regeln zur Namensgebung und zu Zahlenformaten. *Elektronik*, Heft 18/2001, S. 52-59.

[CS2] S. Programmierer: Guter Code braucht Ordnung, Teil 2: Das Code-Layout. *Elektronik*, Heft 8/2002, S. 74-81.

[CS3] A. Schlau: Guter Code braucht Ordnung, Teil 3: Fehlervermeidung bei der Erstellung von C-Programmen. *Elektronik*, Heft 14/2002, S. 66-71.

[DSTD] The Comany, 003720 Design Standards for Embedded Systems Software 2.0

3 Definitions

3.1 Requirement Format and Numbering

The requirements in this document are stated between a requirement number and a requirement delimiter.

The requirement number follows the subsequent syntax rule

R/SRD/<section>/<number>-<issue>

where

<section> is an identifier for the chapter, section, or subsection to which the requirement belongs to

<number> is an incremental number used within each instance of <section>

<issue> is a number indicating in which issue of the specification the requirement was last changed. Initially it is set to ‘1’.

The requirement delimiter is a parallelogram (#).

If a requirement is no longer valid it is replaced by the text “--- deleted”. Requirement numbers may not be removed or reused.

The verb *shall* is used whenever the described practice is compulsory, *should* is used for recommended practices. Sentences without any of the two verbs within a formal requirement are explanations.

3.2 Typographic Conventions

Text in *italic face* represents a term that is defined or has has special meaning within this document. An instance of such a term is not necessarily in italic face, if the context will leave no doubt that the definition is applicable.

Bold face letters emphasise the textual content and improve readability.

Binary numbers are tagged with a subscript ‘b’, like 0101_b.
Hexadecimal numbers are tagged with a subscript ‘h’, like $3FA9_h$.
Numbers without subscripts are to be interpreted as decimals.

3.3 General Technical Terms and Phrases

In the following table a definition of used technical terms and phrases is found.

Term/Phrase	Meaning
arm a watchdog	reset a watchdog timer and start its clock
clear a bit	make a bit zero
disarm a watchdog	stop the clock of a watchdog timer
kick a watchdog	reset the watchdog timer, i.e. reload it with a previously defined value
mask an interrupt	disable interrupt serving but latch the interrupt
set a bit	make a bit one

Table 1: Technical terms used in this document.

dis)arm a watchdog wird nirgends verwendet

4 General Description

4.1 IOPS Environment

The objective of PLS is to provide the power plant designer with a set of sensors that all can communicate with the bus master, the main control unit of the power plant. The Intelligent Oil Pressure Sensor is one of them. The IOPS hosts a mechanical switch that opens if the oil pressure of the turbine lubrication is lower than a temperature dependent threshold, and closes if it is higher. In normal operation the switch should be closed and an open switch would indicate a failure in the lubrication system. However, when the turbine runs on low speed it might open temporarily, and when the turbine stops it must open – otherwise it would mean that the switch is stuck.

The main control unit obtains the status of all sensors at regular intervals via status messages on the PP-bus.

4.2 Purpose and General Functional Principle

After power-up the IOPS gets programmed. The debounce times and report rates are negotiated via the PP-bus. Upon a start command from the main control, the MC4711 starts transmitting the debounced state of the oil pressure switch at the negotiated rate (in the range of seconds). Primarily this messages are sent to show that the IOPS is alive. However, whenever the debounced oil pressure switch state changes, the IOPS immediately sends a state message to the main control (which must start appropriate actions).

During start-up the MC4711 performs a self-test. If it fails, then the IOPS does not respond to any command to show it is broken. In case of errors during the operation (i.e assertion failures) the MC4711 reboots as shown in the state diagram of Figure 1.

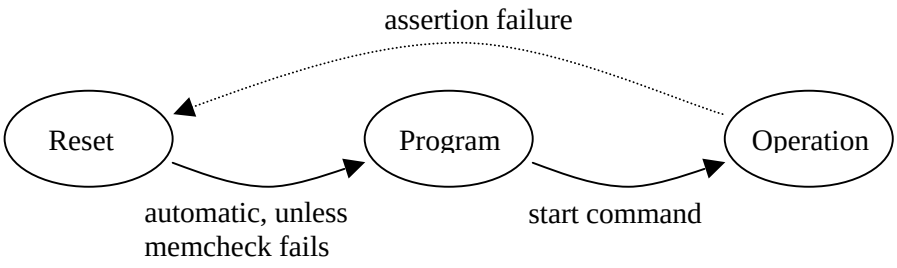


Figure 1: State transitions of the IOPS.

4.3 Hardware

The IOPS is based on the MC4711 Micro Controller, which implements a fully operational PP-bus controller. Thus, issues like bus arbitration, address decoding, and message checksums are fully transparent to the firmware. The MC4711 is a 16-bit controller running at 10MHz with 16kB on-board RAM and 16kB EEPROM. It also contains a digital I/O port. One of its pins is connected to the mechanical switch, one to an external watchdog, and one to a light emitting diode [HW].

5 Specific Software Requirements

This chapter contains the formal requirements on the IOPS software.

5.1 Functional Requirements

5.1.1 Requirements on the Initialisation Process

R/SRD/INIT/10-1: Self Test
After a reset the IOPS software shall perform a selft-test that checks the RAM and the processor core. #

R/SRD/INIT/20-1: Self Test Fail Silence
The IOPS software shall only reply to PP-bus commands when the self-test at start-up passes. #

5.1.2 Requirements on the PP-bus Communication

Each PP-bus message consists of 4 bytes. The first byte is an address byte, it is followed by two *data bytes* and a checksum byte. The first and last bytes are stripped off by the PP-bus controller.

R/SRD/PP/05-2: Ready Message
If the self test (R/SRD/INIT/10) passes, the IOPS software shall send a PP-bus message of the following format.

1st byte	2nd byte
22_h	VER

where VER identifies the firmware version. #

Comment: This message informs the main control about unwanted re-boots of the MC4711 of the IOPS and about the need to re-program the IOPS. The exact format of VER is up to the software design.

R/SRD/PP/10-2: Message Definitions --- deleted

Table 2: --- deleted.

R/SRD/PP/20-1: PP-bus Performance
Upon receipt of a command via the PP-bus, the IOPS shall reply within 1ms (measured between the first byte sent and the last byte received). #

R/SRD/PP/30-1: Reply Message Format
The first *data bytes* of the reply message of the IOPS shall be the ID 23_h. The second byte shall be a copy of the first *data byte* of the command the IOPS is replying to. #

R/SRD/PP/40-1: SetPosDebTime Command
Upon receipt of the data bytes

1st byte	2nd byte
02_h	POS

the IOPS software shall set the positive debounce time for the oil pressure switch to an appropriate offset + POS, where POS shall be interpreted as a signed two-bit complement number and the programmable range shall be from 0 to 2 seconds. #

Comment: the positive debounce time is the debounce time for the transition closed to opened switch.

R/SRD/PP/50-1: SetNegDebTime Command
Upon receipt of the data bytes

1st byte	2nd byte
03_h	NEG

the IOPS software shall set the negative debounce time for the oil pressure switch to an appropriate offset + NEG, where NEG shall be interpreted as a signed two-bit complement number and the programmable range shall be from 0 to 2 seconds. #

Comment: the negative debounce time is the debounce time for the transition opened to closed switch.

R/SRD/PP/60-1: SetRepIntv Command
Upon receipt of the data bytes

1st byte	2nd byte
04_h	SEC

the IOPS software shall report the *debounced oil pressure switch state* at intervals of min(SEC,30) seconds as soon as it has been switched to operational mode by the message of R/SRD/PP/70. #

R/SRD/PP/70-2: Start Command
Upon receipt of the data bytes

1st byte	2nd byte
01_h	XX

the IOPS software shall start reporting the *debounced oil pressure switch state* at intervals specified in R/SRD/PP/60. The value of the 2nd byte of this message is not defined. #

Comment: Theoretically there is no need for a separate Start Command. The IOPS software could transit to operational mode as soon as it receives the SetRepIntv Command. However, this is not wanted to be compatible with other sensors of the LSP.

R/SRD/PP/80-1: Report Message Format
The IOPS shall report the *debounced oil pressure switch state* by sending the following PP-bus message

1st byte	2nd byte
20_h	STC

where STC = 1 in case of an open switch and STC = 0 in case of a closed switch.

Was wenn nach der READY-Nachricht gleich ein START-Kommand kommt? Welche Default-Werte? Wenn das nicht spezifiziert ist und egal ist, dann ins User Manual! Sowas könnte man ggf. durch Aufzeichnen von Szenarios (Sequence Diagrams) erkennen.

5.1.3 Requirements on the Sensor Interface

Low oil pressure opens the oil pressure switch and the LSB of the digital I/O port register of the MC4711 becomes 1. High oil pressure closes the switch, the LSB of the I/O port register becomes zero [HW]. To ease the distinction in the following the term *position* will be used for the physical state of the sensor (the raw sensor value), whereas the debounced switch *state* is what the IOPS reports.

R/SRD/SI/10-1: Switch Position Sampling Rate
The IOPS software shall sample the position of the oil pressure switch at a rate of 1kHz. #

R/SRD/SI/20-2: Switch Position Spikes --- deleted

R/SRD/SI/30-2: Debouncing Algorithm
To compute the *debounced oil pressure switch state* the IOPS software shall use an algorithm that

integrates over the raw sensor values as outlined in Figure 2. If all raw values sampled within the debounce time correspond to a different state than the current debounced oil pressure switch state, then the state flips and it takes at least the same number of samples of opposite switch positions to flip the state again. #

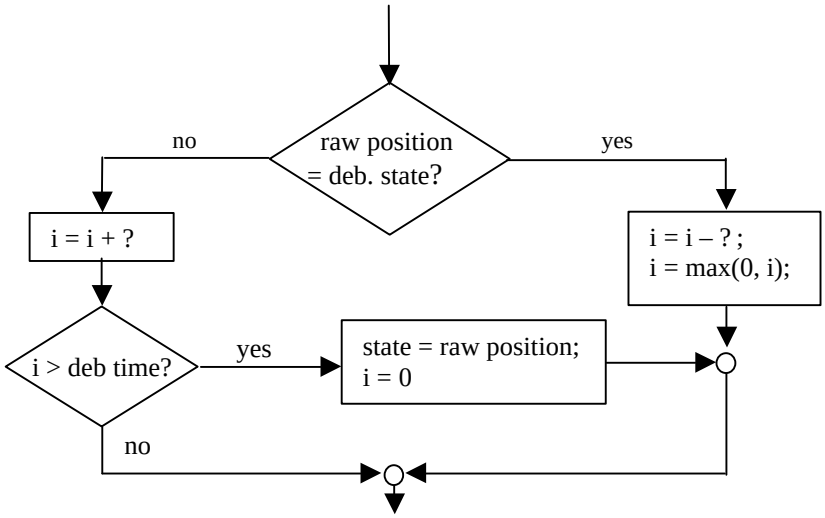


Figure 2:Switch Postition Debouncing.

5.1.4 Reliability and Accuracy

R/SRD/RA/10-2: Missing Parameters
In case the IOPS hasn't received a single valid message within 10 seconds after start-up it shall reboot the processor. #

Comment: The reboot may be initiated by the software by neglecting to kick the watchdog.

R/SRD/RA/20-1: PP-bus Error
In case the software cannot interpret a PP-bus command or encounters an invalid software state, it shall initiate a reboot of the processor within one second. #

Comment: The software should be operable as much as possible, there is no need to log the cause of the error or stay fail silent, once the software is fully operational.

R/SRD/RA/30-1: Watchdog Kicking
The software shall kick a watchdog at least once a second and at most 10000 times a second by setting bit 1 of the digital I/O port for at least 50 clock cycles and by clearing the bit afterwards. #

Comment: the external watchdog will reset the MC4711 if it is not kicked for 1000ms. A deadlock resolved by the watchdog shall not be visible to other devices on the PP-bus.

Das wird ja wohl nicht erfüllt. Siehe FMEA, FTA: bei einem WD reset in operational hört der IOPS auf seinen Zustand zu melden. Eigentlich ist das ein Requirement und kein Kommentar! Was im SRD 1.0 schon bemängelt wurde, hat der SRD-Autor hier eine unsinnig umformliert.

5.2 Software Budget

R/SRD/SB/10-1: Maximum CPU Load
The timing budget of the IOPS software shall load the CPU up to at most 80%. #

R/SRD/SB/20-1: Idle Time
If the MC4711 is not busy with processing, it shall execute the *idle command* to save power and prolong the lifetime of the micro controller. #

Comment: The *idle command* puts the MC4711 in a low-power state. The micro controller proceeds program execution after executing the idle command if and only if an unmasked interrupt has been served since the idle command.

R/SRD/SB/30-1: Memory Footprint
The IOPS software shall consume at most 80% of the MC4711 on-chip memory. #

5.3 Software Design

R/SRD/SD/10-1: Software Codig Standard
The IOPS software shall conform to the software coding standard of [CS1, CS2, CS3]. #

R/SRD/SD/20-1: Software Re-use
The IOPS software shall re-use as much source code from the KJ project as possible. #

R/SRD/SD/30-1: Programming Language
The IOPS software shall be programmed in C whenever possible. #

6 Traceability

This chapter captures the origin of the software requirements in tabular form.

Req#	Req Short Text	Origin
R/SRD/INIT/10	Self Test	[MRD] section 2.1
R/SRD/INIT/20	Self Test Fail Silence	[MRD] section 2.1
R/SRD/PP/05	Ready Message	[MoM] RID 1
R/SRD/PP/20	PP-bus Performance	[ICD] R/ICD/P/030
R/SRD/PP/30	Reply Message Format	[ICD] R/ICD/MSG/110
R/SRD/PP/40	SetPosDebTime Cmd	[ICD] R/ICD/MSG/130
R/SRD/PP/50	SetNegDebTime Cmd	[ICD] R/ICD/MSG/140
R/SRD/PP/60	SetRepIntv Cmd	[ICD] R/ICD/MSG/160
R/SRD/PP/70	Start Command	[ICD] R/ICD/MSG/105
R/SRD/PP/80	Report Message Format	[ICD] R/ICD/MSG/125
R/SRD/SI/10	Switch Position Sampling Rate	minutes of the technical meeting ABB/Thecompany, 12 Apr 2003, Göteborg
R/SRD/SI/30	Debounciung Algorithm	e-mail 0815 from j.r.ewing@dallas.com to arnold@muskelkater.at
R/SRD/RA/10	Missing Parameters	[MRD] section 4
R/SRD/RA/20	PP-bus Error	[MRD] section 4
R/SRD/RA/30	Watchdog Kicking	[HW]
R/SRD/SB/10	Maximum CPU Load	[DSTD]
R/SRD/SB/10	CPU Idle Time	[MRD] section 2
R/SRD/SB/30	Memory Footprint	[DSTD]
R/SRD/SD/10	Software Codig Standard	[DSTD]
R/SRD/SD/20	Software Reuse	decisition of the project management, see TheCompany Mom 0009240, 24 Mar 2003
R/SRD/SD/30-1	Programming Language	for the MC4711 there is no other compiler available

TheCompany Ltd.
Rosentalgasse 3
A-1140 Wien

PLS – Intelligent Oil Pressure Sensor

Architectural Software Design

Document: **004599**
Issue: **1.0**
Date: **15 Sep 2003**

	Name:	Function:	Signature:	Date:
Prepared:	Stephan Grünfelder	SW Designer	……	……..
Checked:	Bernd Spiess	Project Engineer	……	……..
	Robert Stolz	Verification Engineer	……	……..
	Helga Machart	SW Quality Assurance	……	……..
Approved:	Norbert Wichtig	Project Manager	……	……..

Document Distribution List

Hansi Hinterseer, Arnold Schwarzenegger, Cindy Crawford, Hubert von Goisern

Change Log

Version, Date	Change Description
1.0, 15 Sep 2003	new document

This document follows the ESA guidelines PSS-05-04 with minor modifications. © Stephan Grünfelder, stephan.gruenfelder@gmx.at

Table of Contents

1 Intoduction 4
1.1 Purpose 4
1.2 Definitions, Acronyms, Abbreviations 4
2 Documents 4
2.1 Applicable Documents 5
2.2 Reference Documents 5
3 System Overview 5
4 System Context 5
5 System Design 6
5.1 Design Method 6
5.2 Decomposition Description 6
6 Component Description 7
6.1 Debounc 7
6.2 Dio 7
6.3 Messages 8
6.4 Main 9
7 Feasibility and Resource Estimates 10
8 Traceability Matrix 10

This document follows the ESA guidelines PSS-05-04 with minor modifications. © Stephan Grünfelder, stephan.gruenfelder@gmx.at

1 Intoduction

1.1 Purpose

This document describes the architectural design of the software for the Intelligent Oil Pressure Sensor (IOPS). The IOPS is a sensor for the Project Little Stream (PLS), a sensor network for medium sized water power plants.

The intended reader of the document is any manager, software engineer, and verification responsible in the PLS IOPS. To fully understand the contents of this document the reader should be familiar with the software requirements [SRD]. Note that the document describes the design of the IOPS software, only. The design of the overall system (interactions between the subsystems) within a PLS network is found in another document [ICD].

Due to the limited size and complexity of the of IOPS requirements, there exists no detailed design document.

1.2 Definitions, Acronyms, Abbreviations

The IOPS software design method is based on the Unified Modelling Language, which is normally used for OO languages. Hence, it uses OO terminology and this terminology is used in this document, as well. However, since C is not an OO language, a matching for the OO terms is defined as follows:

- a *class* is a C file plus correspondig header file, which are used to implement at least one design item
- an *object* is a class unless we talk about instances of an object
- an *instance* of an object is a `typedef`ed identifier that identifies one usage of a class with specific data
- C-functions operating on data belonging exclusively to a class are called *member functions* of the class

The following acronyms are used in this document:

AD	Applicable Document
CPU	Central Processing Unit
IOPS	Intelligent Oil Pressure Sensor
OO	Object Oriented
PLS	Project Little Stream
RD	Reference Document
SRD	Software Requirements Document

2 Documents

In the event of a conflict between this document and an Applicable Document (AD), the AD shall have the precedence. In the event of a conflict between this document and a Reference Document (RD), this document shall have precedence.

Any such conflict should, however, be brought to the attention of TheCompany Ltd. for resolution.

This document follows the ESA guidelines PSS-05-04 with minor modifications. © Stephan Grünfelder, stephan.gruenfelder@gmx.at

This document has been established based on the issues of ADs and RDs as given below. Issue changes of ADs and RDs will lead to an update of this document only in case of impacts on its content.

The valid documentation status is reflected in the relevant Configuration Items Data List.

2.1 Applicable Documents

[SRD]	TheCompany, 004589	IOPS Software Requirements Document 2.0
[ICD]	ABB, IA-PL-1520-00	PowerPlan-Bus Interface Control Document, Release 1.0

2.2 Reference Documents

[UMLRT]	Bruce Powel Douglass	Real-Time UML, Addison-Wesley, 1998
[MC4711]	Digital Devices	The MC4711 Users Manal, 2nd edition, June 2002
[PMC]	Digital Devices	A Programmer's Guide to the MC4711, 1st edition, October 2002.
[KJADD]	TheCompany, 004403	PLS KJ Software Architectural Design Document 3.0

3 System Overview

The IOPS software is designed to be fail silent, i.e. it rather shuts up before saying something wrong. During start-up a self test of the MC4711 is being performed. If it fails, the software runs an endless loop and waits until the watchdog resets the processor. If the test passes, the IOPS software enables the PP-bus interrupts, sends the PP-bus message `MSG_READY` to inform the main control that it is ready for commands, and starts responding to PP-bus commands from the main control.

At the same time the on-chip timer of the MC4711 is started. The timer is used to generate interrupts at the rate of 1kHz. Its interrupt service routine (ISR) increases the global milliseconds counter `ulSystemTime` by one. This counter is a global variable for reasons of simplicity and is used to implement timeouts and the measurement of the report intervals of the debounced oil pressure switch state.

After the receipt of the command `MSG_START` the IOPS software must no longer listen to PP-bus messages. The IOPS software disables PP-bus interrupts and starts sending its report messages `MSG_OILSWITCH`.

4 System Context

Since the hardware takes care of PP-bus arbitration, address definition, address decoding, checksum calculation, and rejection of messages with wrong checksums, the logical view of the PP-bus communication with the main control is a peer-to-peer communication for the IOPS software [MC4711].

The only other external interfaces of the IOPS software are the oil pressure switch and the external watchdog. Both are connected to the digital I/O port of the controller.

To access the digital I/O port and the PP-bus the MC4711 system C-library is used. It provides the blocking I/O functions `rx_bus()` and `tx_bus()` to receive and send data via the PP-bus, and the functions `memout()` and `memin()` to access special purpose registers of the controller, such as the interrupt configuration register and the digital I/O port register.

This document follows the ESA guidelines PSS-05-04 with minor modifications. © Stephan Grünfelder, stephan.gruenfelder@gmx.at

5 System Design

5.1 Design Method

The used design method is the Unified Modelling Language for Real Time Systems [UMLRT]. This method is an object oriented design method. Since the IOPS software is written in C, the object's attributes are static variables of the source file implementing the class. There is only one instance of each of the identified objects in the IOPS software, which simplifies matters. However, the debouncing algorithm (the class `debounc`) was taken over from the earlier KJ project. This class allows to have multiple instances. To accomplish this, the debouncing state machines are designed as abstract data type `TDebouncer`. A valid instance of this abstract data type has to be passed to each member function, when operating with the debounding algorithm, see section 6.1.

5.2 Decomposition Description

The IOPS software encapsulates the digital I/O port of the MC4711 in the object `Dio`. It has a member function for each peripheral access possible via the port, as seen in the decomposition diagram in Figure 1. This figure only shows a subset of private (i.e. `static`) data and functions, which are identified by a minus as function prefix.

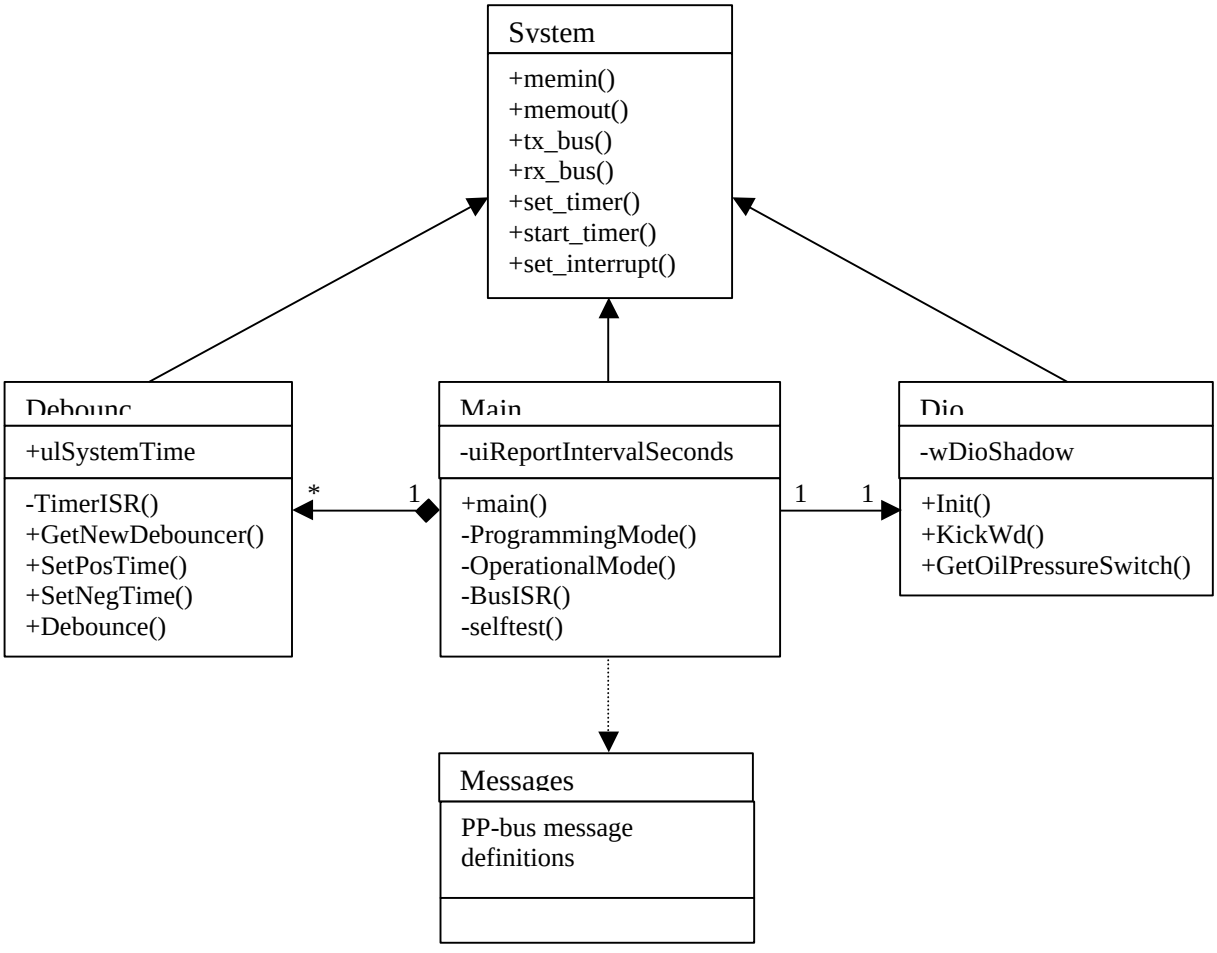


Figure 1: IOPS Software UML Class Diagramm.

This document follows the ESA guidelines PSS-05-04 with minor modifications. © Stephan Grünfelder, stephan.gruenfelder@gmx.at

All processor related functions are found in the `System` library, which is a third party product, and is used by all project specific objects. The object `Debounc` implements a debouncer, i.e. a state machine that debounces digital input. As said only one instance is used by `Main`, which implements the PP-command interpreter and the glue logic for the other objects. Finally, `messages.h` is a header file common to all PLS software projects. It defines the messages sent over the PP-bus.

6 Component Description

6.1 Debounc

6.1.1 Type

The files debounc.c and debounc.h implement a number of identical debouncing functions as abstract data types `TDebouncer`. The maximum number of debouncers that can be handled must be defined with the macro `MAX_DEBOUNCERS`.

6.1.2 Purpose

This component implements the debouncing of the oil pressure switch position, i.e. it is used to compute the debounced oil pressure switch state. Furthermore it keeps the system time, i.e. increments the global variable `ulSystemTime` at 1kHz with the help of the on-chip timer.

6.1.3 Function

The function `GetNewDebouncer()` returns a handle of type `TDebouncer` for a new debouncer and initialises the private variables and the on-chip timer in case the function is called the first time.

The member functions `SetPosTime()` and `SetNegTime()` set the positive and negative debounce times of a debouncer and at the same time define the initial state of the debouncer.

The function `Debounce()` takes as argument the raw digital sensor value and returns the debounced value. The algorithm exaclty follows the diagram shown in figure 2 of the [SRD] with ? = 1ms. The state of the debouncer(s) is captured in arrays, which have the dimension `MAX_DEBOUNCERS`.

6.1.4 Subordinates

To set-up the 1kHz timer interrupt the MC4711 system library is used.

6.1.5 Dependencies

Before any of the other member functions can be called, a valid debouncer must be created by calling `GetNewDebouncer()`.

6.1.6 Interfaces

The member functions only accept and return built-in types.

6.1.7 Resources

Debounc needs the on-board timer and defines an interrupt service routine for the cyclic timer expiration.

6.2 Dio

6.2.1 Type

The component Dio is a high level driver for all digital I/O of the IOPS software.

This document follows the ESA guidelines PSS-05-04 with minor modifications. © Stephan Grünfelder, stephan.gruenfelder@gmx.at

6.2.2 Purpose

The idea of Dio is to ease the use of the digital I/O port of the MC4711. Instead of accessing the pins of the port in the client routine, the knowledge which pin is connected to which peripheral device is encapsulated in Dio.c.

6.2.3 Function

The member function `Init()` sets up the input/output configuration of the port. All other member functions are project specific getter and setter methods that access the pins of the port. When writing to the port register the MC4711 demands that the value last read from an input pin has to be written [MC4711]. This is accomplished in Dio by keeping a shadow register which is updated it in the getter functions and used in the setter functions.

6.2.4 Subordinates

To access the digital I/O port the system library functions are used.

6.2.5 Dependencies

`Init()` must be called before any other member function is called.

6.2.6 Interfaces

The member functions only accept and return built-in types.

6.2.7 Resources

No other software component should access the digital I/O port in parallel. This could cause wrong values in the port shadow register.

6.3 Messages

6.3.1 Type

The component is a header file, only.

6.3.2 Purpose

The idea of messages.h is to share a file in all PLS software projects. This file defines all PP-bus messages in a central place.

6.3.3 Function

The file only contains #defines, but no enumerations. This is to allow to #include this file in Assembler programs, as well.

6.3.4 Subordinates

No subordinates.

6.3.5 Dependencies

See above.

6.3.6 Interfaces

See above.

6.3.7 Resources

No resources are used.

This document follows the ESA guidelines PSS-05-04 with minor modifications. © Stephan Grünfelder, stephan.gruenfelder@gmx.at

6.4 Main

6.4.1 Type

Apart from `main()`, Main has no public member functions. Hence, it has no header file.

6.4.2 Purpose

Main implements the big bulk of software requirements on the IOPS. Virtually all except the debouncing and the low level routines.

6.4.3 Function

Main implements the following state machine. The text at the transition edges indicates what happens inside main.c.

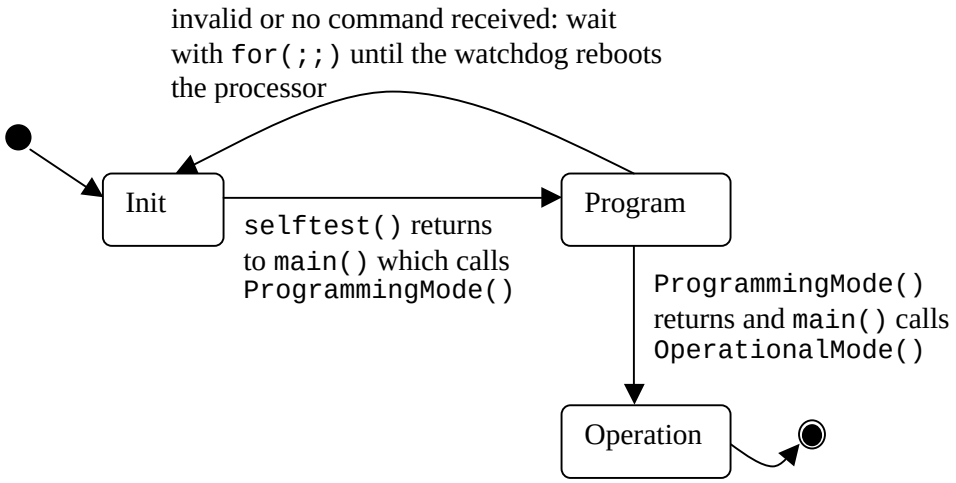

invalid or no command received: wait with for(;;) until the watchdog reboots the processor
Init
selftest() returns to main() which calls ProgrammingMode()
Program
ProgrammingMode() returns and main() calls OperationalMode()
Operation

Figure 2: UML state chart for the IOPS software.

The `main()` function initialises the hardware and called components, calls `selftest()`, sends the message `MSG_READY` to the main control. After that it passes the control to `ProgrammingMode()`and `OperationalMode()`.

The private member function `ProgrammingMode()` interprets and dispatches the PP-bus commands, and replies to them. If a syntactically correct but semantically unkown command is received, it stops kicking the watchdog. The same happens if no valid command has been received since start-up for 10 seconds.

To be able to respond to the PP-bus commands in time and kick the watchdog, `ProgrammingMode()` enables the PP-bus receive-interrupt and the timer interrupt. The function `idle()` is called in a loop. After this call `ProgrammingMode()`checks which interrupt has woken up the CPU.

The private member function `OperationalMode()` disables all interrupts but the timer interrupt. It contains an endless loop that waits with `idle()` until the timer interrupt comes, gets the switch position with the help of Dio and processes this data with a `TDebouncer`. It sends `MSG_OILSWITCH` messages at the required intervals. The interval length is measured with the help of `ulSystemTime`.

The self test is a takeover from the KJ project. For its design refer to [KJADD]

6.4.4 Subordinates

Main uses a `TDebouner`, the Dio, and the system library.

This document follows the ESA guidelines PSS-05-04 with minor modifications. © Stephan Grünfelder, stephan.gruenfelder@gmx.at

6.4.5 Dependencies

The only public function in Main is `main()`. It must be the first C-function called by the runtime-environment.

6.4.6 Interfaces

`main()` does not expect any parameters.

6.4.7 Resources

The object hosts an empty PP-bus Receive Interrupt service routine and requires an extra program memory segment for the self test.

7 Feasibility and Resource Estimates

From experience with similar projects it is estimated that less than 25% of the on-chip memory are used and the CPU is idle 92% of the time.

8 Traceability Matrix

The following table depicts the trace form software requirements to the C-function implementing the requirements.

Req#	Req Short Text	Top Level Functionality Implemented in
R/SRD/INIT/10	Self Test	Main.selftest
R/SRD/INIT/20	Self Test Fail Silence	Main.selftest
R/SRD/PP/05	Ready Message	Main.main
R/SRD/PP/20	PP-bus Performance	Main.ProgrammingMode
R/SRD/PP/30	Reply Message Format	Main.ProgrammingMode
R/SRD/PP/40	SetPosDebTime Cmd	Main.ProgrammingMode
R/SRD/PP/50	SetNegDebTime Cmd	Main.ProgrammingMode
R/SRD/PP/60	SetRepIntv Cmd	Main.ProgrammingMode
R/SRD/PP/70	Start Command	Main.ProgrammingMode, Main.OperationalMode
R/SRD/SI/10	Switch Position Sampling Rate	Debounc.GetNewDebouncer, Main.OperationalMode
R/SRD/SI/30	Debouncing Algorithm	Debounc.Debounce
R/SRD/RA/10	Missing Parameters	Main.ProgrammingMode
R/SRD/RA/20	PP-bus Error	Main.ProgrammingMode
R/SRD/RA/30	Watchdog Kicking	Main.ProgrammingMode, Main.OperationalMode
R/SRD/SB/10	Maximum CPU Load	not a functional requirement
R/SRD/SB/10	CPU Idle Time	not a functional requirement
R/SRD/SB/30	Memory Footprint	not a functional requirement
R/SRD/SD/10	Software Codig Standard	not a functional requirement
R/SRD/SD/20	Software Reuse	not a functional requirement
R/SRD/SD/30-1	Programming Language	not a functional requirement

This document follows the ESA guidelines PSS-05-04 with minor modifications. © Stephan Grünfelder, stephan.gruenfelder@gmx.at

Traceability Matrix for the Exercise

Req#	Req Short Text	Tested in
R/SRD/INIT/10	Self Test	ST-IOPS-FT01
R/SRD/INIT/20	Self Test Fail Silence	ST-IOPS-FT01
R/SRD/PP/05	Ready Message	ST-IOPS-FT01
R/SRD/PP/20	PP-bus Performance	ST-IOPS-FT02
R/SRD/PP/30	Reply Message Format	ST-IOPS-FT02
R/SRD/PP/40	SetPosDebTime Cmd	ST-IOPS-FT03
R/SRD/PP/50	SetNegDebTime Cmd	ST-IOPS-FT03
R/SRD/PP/60	SetRepIntv Cmd	ST-IOPS-FT02
R/SRD/PP/70	Start Command	ST-IOPS-FT02
R/SRD/PP/80	Report Message Format	ST-IOPS-FT02
R/SRD/SI/10	Switch Position Sampling Rate	ST-IOPS-FT03
R/SRD/SI/30	Debouncing Algorithm	ST-IOPS-FT04, ST-IOPS-FT03
R/SRD/RA/10	Missing Parameters	ST-IOPS-FT05
R/SRD/RA/20	PP-bus Error	ST-IOPS-FT05
R/SRD/RA/30	Watchdog Kicking	ST-IOPS-FT05
R/SRD/SB/10	Maximum CPU Load	ST-IOPS-FT06
R/SRD/SB/20	CPU Idle Time	ST-IOPS-FT07
R/SRD/SB/30	Memory Footprint	ST-IOPS-FT08
R/SRD/SD/10	Software Codig Standard	tested in code reviews
R/SRD/SD/20	Software Reuse	tested in code reviews
R/SRD/SD/30-1	Programming Language	tested in code reviews

Medcare

[Projectname, Test Object]

Software Test Plan

Copyright © 2004 Medcare hf. All Rights Reserved

Document Number: D-0209-00
Document Revision: 1.0
Date: 3 Sep 2002

Document Classification: CONFIDENTIAL
This document contains proprietary confidential information that belongs exclusively to Mecare hf., IS-105 Reykjavík, Austurhlið 7. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form, or by any means, electronic, mechanical, optical, chemical, manual, or otherwise, without the prior written authorization from Medcare hf.

PREPARED, CHECKED	FUNCTION:	SIGNATURE:	DATE:
[Not the SW Designer]	Test Design	.....	.....
[The SW Designer]	Software Design	.....	.....
[Sys. Knowledge Person]	System Engineer	.....	.....
Stefan Stefansson	Project Manager	.....	.....

Document Distribution

[write here all persons involved in the project and let the project manager approve this list]

Document Change Log

Issue, Date	Modified Section	Change Description
1.0, 3 Sep 2002	all	initial version, Sepp Hintergruber

Table of Contents

1 Introduction 3
1.1 Purpose 3
1.2 Background 3
1.3 Scope 3
1.4 Acronyms 3
2 Requirements for Test 4
3 Test Strategy 5
3.1 Testing Types 5
3.1.1 Data Integrity Testing 5
3.1.2 Function Testing 6
3.1.3 Business Cycle Testing 8
3.1.4 User Interface Testing 9
3.1.5 Performance Profiling 10
3.1.6 Load Testing 11
3.1.7 Stress Testing 11
3.1.8 Volume Testing 12
3.1.9 Input Range Testing 12
3.1.10 Security and Access Control Testing 12
3.1.11 Failover and Recovery Testing 13
3.1.12 Configuration Testing 14
3.1.13 Installation Testing 14
3.1.14 Software Build Consistency Check 15
3.2 Test Tools 15
4 Resources 16
4.1 Roles 16
4.2 System 17
5 Project Milestones 18
5.1 Milestones Plan 18
5.2 Progress Assessment 18
6 Deliverables 19
6.1 Test Model 19
6.2 Test Logs 19
6.3 Defect Reports 19
7 References 20
7.1 Applicable Documents 20
7.2 Reference Documents 20

1 Introduction

[This is a Software Test Plan Template based on the RUP template [R2] for Software Test Plans. It is, however, stricter than RUP and more detailed; i.e. in RUP the test case description lacks detail.]

1.1 Purpose

This document describes the planned test activity of the [Project, Test Object].

[Identify existing project information and the software components that should be tested, e.g. refer to the requirements document. Describe the basic testing strategies to be employed (e.g. black box SW/SW interface, SW/HW integration testing with oscilloscope, ...).]

Standard resources needed to perform these tests are defined in chapter 4, special test tools are identified in section 3.2.

1.2 Background

[Enter a brief description of the target-of-test (components, application, system, etc.) and its goals. Include information such as major functions and features, its architecture, and a brief history of the project. This section should be not longer than three to five paragraphs.]

1.3 Scope

[Describe the stages of testing covered in this document — for example, Linting, Code Reviews, Unit-, Integration-, or System-Testing — and the types of testing that will be addressed by this plan, such as Function or Performance.

Provide a brief list of the target-of-test's features and functions that will or will not be tested.

List any assumptions made during the development of this document that may impact the design, development or implementation of testing.

List any risks or contingencies that may affect the design, development or implementation of testing.

List any constraints that may affect the design, development or implementation of testing]

1.4 Acronyms

ADC	Analog/Digital Converter
HW	Hardware
IDE	Integrated Development Environment
I/O	input/output
LED	Light Emitting Diode
RTOS	Real Time Operating System
RUP	Rational Unified Process
SW	Software
TBC	To Be Confirmed
TBD	To Be Defined
UI	User Interface
XP	Extreme Programming

2 Requirements for Test

This chapter lists all applicable software requirements and identifies the test case verifying the correct implementation of the requirement in form of a tracablility matrix. In case a requirement is not tested, the right hand column of the following table states “not tested”.

[In case more than one input document is at hand (i.e. a system specification and a interface specification, then create two tables. The tables should preferably be extracted from tools such as DOORS, Requesite Pro).

Note that this is a boring thing to do but the Matrix’ careful creation, review, and update is an essential part in the proof that really all requirements are implemented and tested. The complementary part in this proof are the test procedures itself. That is why the matrix and the test procedures reside in the same document.

An example matrix is given below.

SW Requirement	Test of the Requirement
R-CI-10: Device Init Status	ST-CUDSP-FT01
R-CI-20: Shutdown Message	ST-CUDSP-FT02
R-MH-10: Button Debouncing	ST-CUDSP-FT01, ST-CUDSP-FT02
R-GUI-15: Window appearance	ST-CUDSP-FT03
…	…

NOTE: In case you do module testing (white box function level testing), or SW/SW integration testing, then just describe the the test strategy (i.e. use of a module test tool, compiler settings for the test environment) in the next chapter and refer to the test cases which should be in electronic form, only.]

3 Test Strategy

[The Test Strategy presents the recommended approach to the testing of the target-of-test. The previous section, Requirements for Test, described what will be tested — this describes how the target-of-test will be tested.

For each type of test, provide a description of the test and a short description what is covered with the test.

If a type of test will not be implemented and executed, indicate this in a sentence stating the test will not be implemented or executed and stating the justification, such as “This test will not be implemented or executed. This test is not appropriate.”

The main considerations for the test strategy are the techniques to be used and the criterion for knowing when the testing is completed.

In addition to the considerations provided for each test below, testing should only be executed using known, controlled databases in secured environments.

IMPORTANT: When defining tests try to do them as automatic as possible and as mean as possible. The objective of the test design must be to *break* the test object, not to show that it works fine.]

3.1 Testing Types

[In case the test object has some test features (test signal generators etc.) that have impact on the subsequently described test cases, mention them here briefly.

Every test gets a unique ID which should be used in the file names for automatic tests. The ID consists of a subsequently explained prefix, a hyphen, a test object identifier, and a test number that includes the type of the test. For example ST-CUDSP-FT02 identifies the functional system test number two for the Digital Signal Processor in the Communication Unit of the PowerEmbla.]

3.1.1 Data Integrity Testing

[The databases and the database processes should be tested as a subsystem within the project. These subsystems should be tested without the target-of-test’s User Interface as the interface to the data. Additional research into the DataBase Management System (DBMS) needs to be performed to identify the tools and techniques that may exist to support the testing identified below.]

Identification:	ST-[Test Object Acronym]-IT01
Test Objective:	[Ensure database access methods and processes function properly and without data corruption.]
Technique:	[Invoke each database access method and process, seeding each with valid and invalid data or requests for data. Inspect the database to ensure the data has been populated as intended, all database events occurred properly, or review the returned data to ensure that the correct data was retrieved for the correct reasons. You should specify the]
Completion Criteria:	[All database access methods and processes function as designed and without any data corruption.]
Special Considerations:	[Testing may require a DBMS development environment or drivers to enter or modify data directly in the databases. Best perform the tests automatically: in this way regression tests are accelerated, the manual inspection of data is not needed, and more realistic chunks of data can be used for testing.]

[Such tests also apply to firmware. Even though there is no “database” it produces data and the integrity can be tested. For example

Identification:	ST-CUDSP-IT02
Test Objective:	Make sure the channel data is tagged with the right ID codes.
Technique:	Set the ADC to test mode by shortcutting the test pins on blabla. Start a recording via the command START_REC. Set the system to the highest data sampling possible. And change the sampling rate during the recording from highest to lowest rates at least twice. For each data packet delivered by the test object verify that the channel ID is identical to the channel when being interpreted as unsigned 16-bit words.
Completion Criteria:	All verifications described above pass.
Special Considerations:	Automatic test with a manual set-up.

→ for this section think of tests that might destroy the integrity of the data (obscure combination of data accesses, obscure sizes, ...).]

3.1.2 Function Testing

[Function testing of the target-of-test focuses on the requirements for test that can be traced directly to use cases or system requirements. The goals of these tests are to verify proper data acceptance, processing, and retrieval, and the appropriate implementation of the requirements. This type of testing is based upon black box techniques; that is verifying the application and its internal processes by interacting with the application via its interfaces and analyzing the output or results.

This section will comprise the biggest part of the document. In case of pairing in the sense of XP this section might just refer to the directory of the test cases, which must be reviewed by a pair programmer and the organization of the files there. When you pair it is recommended to first agree on a mockup of the test before implementing the test.

IMPORTANT: If you do not pair then describing the test in textual form is the only way to review the test design. Note that a textual description might be the only way to let

a systems engineer (a hardware engineer, an application expert) to review the test design. The existence of the textual description does not mean a test review does not have to take place.

What follows is a generic test description adapted from RUP:]

Identification:	ST-[Test Object Acronym]-FT01
Test Objective:	[Ensure proper target-of-test functionality, including navigation, data entry, processing, and retrieval.]
Technique:	[Execute each use case, use-case flow, or function, using valid and invalid data, to verify the following: • The expected results occur when valid data is used. • The appropriate error or warning messages are displayed when invalid data is used. • Each business rule is properly applied.]
Completion Criteria:	[In case Verifications described above do not pass, describe any tolerances here.]
Special Considerations:	[Identify or describe those items or issues (internal or external) that impact the implementation and execution of function test]

[Where RUP defines “business rules” as a declaration of policy or condition that must be satisfied within the business. Business rules are kinds of requirements on how the business must operate. They can be laws and regulations imposed on the business, but also express the chosen business architecture and style.

The test plan you write should now create instances of this class of tests covering all the requirements of the Software Requirements Specification. Try to make the tests

- **orthogonal**, i.e. when one requirement changes take care that not all tests have to be revised
- **basic**, i.e. do not test things that are not required, do not invent pass/fail criteria if they are not properly defined
- **mean**, i.e. when you design the tests your intention must be to find bugs, to crash the system, not to proof that everything works smoothly; from possible input ranges of data driven tests always take an invalid input one on the border an valid input on the border, and its neighbour. For example if an algorithm has to work on integers from 0 to 100 test the algorithm with inputs –1, 0, 1, 99, 100, 101
- **automatic**: from the last item it is already clear that manual testing gets boring and is prone to errors; be creative to find means to do tests fully automatically. This also may include the creation of new requirements to improve testability of the system!

An example test design is given below. We assume that the software allows the user (or another system via a command interface) to select an unsigned 16-bit integer output range of a signal. In case the signal amplitude does not fit within this range, it must be clipped. The example text contains reference numbers in parentheses, which refer to comments given later.

Identification:	ST-CUDSP-FT02
Test Objective:	Verify that the selectable signal clipping works.
Technique:	Switch the signal source to test ramp mode. (1) Select an output range from 0 to 0. (2) Process the signal for 2 seconds, verify that the output is a zero line, with the correct number of samples. (3) Select an output range from 0 to 1. Process the signal for 2 seconds, verify that the output is zero sample followed by ones. Verify that the number of samples is correct. Select an output range from 100 to 200. Process the signal for 0.5 seconds, verify that the output comprises 100 zero samples followed by a ramp from 100 to 200 and samples of the value 200. Verify that the number of samples is correct. (4) Perform similar tests for output ranges (65534 to 65535) and (65535 to 65535). Select reversed input ranges, i.e. from 1 to 0 and from 65535 to 1, and process the signal for 2 seconds with these selections. (2)
Completion Criteria:	All verifications described above pass. The reversed input ranges shall result in an error log but shall not stop the system, i.e. at least a correct number of output samples must be given.
Special Considerations:	The software under test is run in batch mode to allow the test to be run automatically. However, the signal source must be switched manually to ramp mode (to be fixed in release 2.0). (6)

(1) We assume that the input signal can be switched to a ramp function covering all possible input values within 1 second. If such a test function can help to automate a lot of tests, it is worth adding such a testability requirement for the device/software providing the input. The implementation of this additional requirement will pay off by making testing easier and faster.

(2) trying to be mean

(3) Two seconds to be sure we cover all possible input values. The counting of the number of clipped samples ensures that there is no algorithmic error suppressing samples instead of clipping them. Doing such pendantic tests by hand is dead boring and error prone.

(4) Do not forget to test a typical range, as well.

(5) We are dealing with integers, no tolerances or whatsoever apply.

(6) The location of the test script(s) executing the automatic test must be defined somewhere in the document at hand. Test scripts can be C-Programs, bat-files, scripts from GUI testing-recorders, Matlab-scripts, or whatever. But stick to one technique: a comfortable test environment should allow to execute a bunch of tests by a single user interaction.

Note: the test design omits out-of-range test cases. In case there is the possibility to define an invalid input range (i.e. from –1 to 1 in a GUI) this must be tested!]

3.1.3 Business Cycle Testing

[Business Cycle Testing should emulate the activities performed on the Project over time. A period should be identified, such as one year, and transactions and activities

that would occur during a year's period should be executed. This includes all daily, weekly, and monthly cycles and, events that are date-sensitive, such as ticklers.

The test design template given below is straight out of RUP. It describes the whole class of tests covered in this subsection and should give you ideas to design your own business cycle tests.]

Identification:	ST-[Test Object Acronym]-BC01
Test Objective:	[Ensure proper target-of-test and background processes function according to required business models and schedules.]
Technique:	[Testing will simulate several business cycles by performing the following: • The tests used for target-of-test's function testing will be modified or enhanced to increase the number of times each function is executed to simulate several different users over a specified period. • All time or date-sensitive functions will be executed using valid and invalid dates or time periods. • All functions that occur on a periodic schedule will be executed or launched at the appropriate time. • Testing will include using valid and invalid data to verify the following: the expected results occur when valid data is used, the appropriate error or warning messages are displayed when invalid data is used. • Each business rule is properly applied.]
Completion Criteria:	[All planned tests have been executed. All identified defects have been addressed.]
Special Considerations:	[System dates and events may require special support activities. Business model is required to identify appropriate test requirements and procedures.]

3.1.4 User Interface Testing

User Interface (UI) testing verifies a user's interaction with the software. The goal of UI testing is to ensure that the User Interface provides the user with the appropriate access and navigation through the functions of the target-of-test. In addition, UI testing ensures that the objects within the UI function as expected and conform to corporate or industry standards.

Do not place tests into this subsection that check functional requirements on the user interface. These tests belong into section 3.1.2. Instead use this section to uncover implicit requirements that were forgotten to be included into the Software Requirements Specification and check the usability of the product and the robustness of the UI.

Identification:	ST-[Test Object Acronym]-UI01
Test Objective:	[Navigation through the target-of-test properly reflects business functions and requirements, including window-to-window, field-to-field, and use of access methods (tab keys, mouse movements, accelerator keys. Window objects and characteristics, such as menus, size, position, state, and focus conform to standards.]
Technique:	[Create or modify tests for each window to verify proper navigation and object states for each application window and objects.]
Completion Criteria:	[Each window successfully verified to remain consistent with benchmark version or within acceptable standard.]
Special Considerations:	[Not all properties for custom and third party objects can be accessed.]

3.1.5 Performance Profiling

[Performance profiling is a performance test in which response times, transaction rates, and other time-sensitive requirements are measured and evaluated. The goal of Performance Profiling is to verify performance requirements have been achieved. Performance profiling is implemented and executed to profile and tune a target-of-test's performance behaviors as a function of conditions such as workload or hardware configurations.]

Identification:	ST-[Test Object Acronym]-PP01
Test Objective:	[Verify performance behaviors for designated transactions or business functions under the conditions of normal anticipated workload and anticipated *worst case workload*]
Technique:	[Use Test Procedures developed for Function or Business Cycle Testing. Modify data files to increase the number of transactions or the scripts to increase the number of iterations each transaction occurs. Scripts should be run on one machine (best case to benchmark single user, single transaction) and be repeated with multiple clients (virtual or actual, see Special Considerations below).]
Completion Criteria:	[For Real-Time Systems the requirements on the performance should be well defined. Other systems might soft criteria as the RUP defines: Single Transaction or single user: Successful completion of the test scripts without any failures and within the expected or required time allocation per transaction. Multiple transactions or multiple users: Successful completion of the test scripts without any failures and within acceptable time allocation.]
Special Considerations:	[For client/server based systems: comprehensive performance testing includes having a background workload on the server. There are several methods that can be used to perform this, including: Create “virtual” user load to simulate many clients, usually several hundred. Remote Terminal Emulation tools are used to accomplish this load. This technique can also be used to load the network with “traffic”. Use multiple physical clients, each running test

	scripts to place a load on the system. For embedded real-time systems you should consider ways to measure the performance, e.g. toggling a pin and recording its status with a digital storage oscilloscope or measuring idle cycles in a RTOS. This might impose testability requirements on the software and hardware of the test object!]

3.1.6 Load Testing

[RUP defines Load testing as a performance test which subjects the target-of-test to **varying workloads** to measure and evaluate the performance behaviors and ability of the target-of-test to continue to function properly under these different workloads. The goal of load testing is to determine and ensure that the system functions properly **beyond the expected maximum workload**.

For the test ID use LT as code that identifies Load Tests.]

3.1.7 Stress Testing

[Stress testing is a type of performance test implemented and executed to find errors due to low resources or competition for resources. Low memory or disk space may reveal defects in the target-of-test that aren't apparent under normal conditions. Other defects might result from competition for shared resources like database locks or network bandwidth. Stress testing can also be used to identify the peak workload the target-of-test can handle.

For PC-based software think of stressing the system by (simultaneously)

- multiple users performing the same transactions against the same data or accounts
- lowering RAM to least possible specified
- filling up the harddisk
- do the worst case transaction volume or mix

For embedded software think of

- a (nearly) full Flash Memory
- interrupts coming at highest rates
- excessive service request or data rates

For stressing and load testing of network based applications a lot of tools exist.

For embedded systems it is recommended to regard memory tests (i.e. stack size) as special stress tests, since memory could be regarded as a resource. The following example is such a test taken from the Embla N/S7000 CU DSP Software Test Plan.

Identification:	ST-CUDSP-ST01
Test Objective:	Check that the stack space reserved is sufficient.
Technique:	Start the CU DSP Software via a Linux Terminal. Start the utility EDSPMon. Fill the top half of the stack with a nonzero pattern, i.e. 0xDEAD. (The syntax of the filling command `dmm` is obtained with the `help` command. The stack address is found when looking for `seg_stack` in the link description file). Exit EDSPMon. Perform a recording with an Impedance Test. Start EDSPMon, make a dump of the memory region filled before.
Completion Criteria:	The dump file contains only words of the defined pattern. This means that at most 50% of the stack space is used in normal operation.
Special Considerations:	Manual test.]

3.1.8 Volume Testing

[Volume Testing subjects the target-of-test to large amounts of data to determine if limits are reached that cause the software to fail. Volume Testing also identifies the continuous maximum load or volume the target-of-test can handle for a given period. For example, if the target-of-test is processing a set of database records to generate a report, a Volume Test would use a large test database and check that the software behaved normally and produced the correct report.

For the ID of Volume Test Descriptions use the acronym VT.]

3.1.9 Input Range Testing

[This is a bit validation testing. Check if the results for strange input deliver still useful results (despite that you tested the full range on algorithmic level in unit tests). For example a respiratory effort sensor is designed for patients with up to 1.50m of thorax circumfence. What if there is an extremely big person using the sensor? Can the software still deliver useful results (i.e. do filter coefficients and algorithms need adaptation?)

For the ID of Input Range Test Descriptions use the acronym RT.]

3.1.10 Security and Access Control Testing

[Security and Access Control Testing focus on two key areas of security:

- Application-level security, including access to the Data or Business Functions
- System-level Security, including logging into or remote access to the system.

Application-level security ensures that, based upon the desired security, actors are restricted to specific functions or use cases, or are limited in the data that is available to them. For example, everyone may be permitted to enter data and create new accounts, but only managers can delete them. If there is security at the data level, testing ensures that" user type one" can see all customer information, including financial data, however," user two" only sees the demographic data for the same client.

System-level security ensures that only those users granted access to the system are capable of accessing the applications and only through the appropriate gateways.

When designing tests, create tests for each user type and verify each permission by creating transactions specific to each user type. Modify user type and re-run tests for

same users. In each case, verify those additional functions or data are correctly available or denied.

Tests on the system level might require to change configurations in the operating system or network interface. Therefore system level tests might have to be manual tests.

For the ID of Security and Access Control Test Descriptions use the acronym SA.]

3.1.11 Failover and Recovery Testing

[Failover and RecoveryTesting ensures that the target-of-test can successfully failover and recover from a variety of hardware, software or network malfunctions with undue loss of data or data integrity.

Failover testing ensures that, for those systems that must be kept running, when a failover condition occurs, the alternate or backup systems properly “take over” for the failed system without loss of data or transactions.

Recovery testing is an antagonistic test process in which the application or system is exposed to extreme conditions, or simulated conditions, to cause a failure, such as device Input/Output (I/O) failures or invalid database pointers and keys. Recovery processes are invoked and the application or system is monitored and inspected to verify proper application, or system, and data recovery has been achieved.

Use the following generic test description for ideas to create your own tests.

Identification:	ST-[Test Object Acronym]-FR01
Test Objective:	[Verify that recovery processes (manual or automated) properly restore the database, applications, and system to a desired, known, state. The following types of conditions are to be included in the testing: • power interruption to the client • power interruption to the server • communication interruption via network servers • interruption, communication, or power loss to controllers • incomplete cycles (data filter processes interrupted, data synchronization processes interrupted). • invalid database pointer or keys invalid or corrupted data in database (flash memory, ...)]
Technique:	[Tests created for Function and Business Cycle testing should be used to create a series of transactions. Once the desired starting test point is reached, the following actions should be performed, or simulated, individually: • Power interruption to the client: power the CPU down. • Power interruption to the server: simulate or initiate power down procedures for the server. • Interruption via network servers: simulate or initiate communication loss with the network (physically disconnect communication wires or power down network servers or routers. • Interruption, communication, or power loss to controllers: simulate or physically eliminate communication with one or more controllers or devices. Once the above conditions or simulated conditions are achieved, additional transactions should be executed and upon reaching this second test point state, recovery procedures should be invoked.

	Testing for incomplete cycles utilizes the same technique as described above except that the database processes themselves should be aborted or prematurely terminated. Testing for the following conditions requires that a known database state be achieved. Several database fields, pointers, and keys should be corrupted manually and directly within the database (via database tools). Additional transactions should be executed using the tests from Application Function and Business Cycle Testing and full cycles executed.]
Completion Criteria:	[In all cases above, the application, database, and system should, upon completion of recovery procedures, return to a known, desirable state. This state includes data corruption limited to the known corrupted fields, pointers or keys, and reports indicating the processes or transactions that were not completed due to interruptions.]
Special Considerations:	[Recovery testing is highly intrusive. Procedures to disconnect cabling (simulating power or communication loss) may not be desirable or feasible. Alternative methods, such as diagnostic software tools may be required. Resources from the Systems (or Computer Operations), Database, and Networking groups are required. These tests would normally run after office hours or on an isolated machine.]

3.1.12 Configuration Testing

[Configuration testing verifies the operation of the target-of-test on different software and hardware configurations. In most production environments, the particular hardware specifications for the client workstations, network connections and database servers vary. Client workstations may have different software loaded — for example, applications, drivers, etc. — and at any one time, many different combinations may be active using different resources.

When designing tests think of interactions with software that is not a target-of-test. Are there different versions that must be supported or configuration changes that might influence the test result?

For the ID of Configuration Test Descriptions use the acronym CT.]

3.1.13 Installation Testing

[Installation testing has two purposes. The first is to insure that the software can be installed under normal and abnormal conditions. Abnormal conditions include insufficient disk space, lack of privilege to create directories, etc. The second purpose is to verify that, once installed, the software operates correctly. This usually means running a number of the tests that were developed for Function Testing.

For test design consider to verify, that the target-of-test properly installs onto each required hardware configuration under the following normal conditions:

- new installation, a new machine, never installed previously
- update, machine previously installed, same version
- update, machine previously installed, older version

On the installed version run a defined subset of function tests to verify the correct installation.

For the ID of Configuration Test Descriptions use the acronym IN.]

3.1.14 Software Build Consistency Check

[RUP has no test reserved to check if the tested software build is actually the one intended to test. But this document has. The tests addresses the execution of a quality assurance policy.]

Identification:	ST-CUDSP-SB01
Test Objective:	Check the consistency of the labeled source code in the version control system and the object-of-test.
Technique:	On a clean PC: install the compiler, get the software sources of the target-of-test, follow the build instructions for the software.
Completion Criteria:	The resulting executable is the target-of-test.
Special Considerations:	The target-of-test should also be found in the version control system. Consider to make a snapshot of source directories and store them separately (out of the version control system).

3.2 Test Tools

The following tools will be employed for the testing campaign.

[Think that test tools could be test data generators, network load generators, defect racking tools, profilers, emulators, oscilloscopes, and many more. It is necessary to plan what is needed at what time (needs to be purchased, needs to be developed) to avoid surprises when the project runs out of time!]

Tool	Vendor	Version

4 Resources

4.1 Roles

Human Resources		
Role	**Persons (part/full time)**	**Responsibilities, Comments**
Test Manager	[Assign names here in this column! More than one roles can be taken by one and the same person.]	Provides management oversight. Responsibilities: • provide technical direction • acquire appropriate resources • provide management reporting • conduct test reviews
Test Designer		Identifies, prioritizes, and implements test cases. Responsibilities: • design tests • evaluate effectiveness of test effort
Tester		Executes the tests. Responsibilities: • execute tests • log results • recover from errors • document change requests
Test System Administrator		Ensures test environment and assets are managed and maintained. Responsibilities: • administer test management system • install and manage access to test systems
Database Administrator, Database Manager		Ensures test data (database) environment and assets are managed and maintained. Responsibilities: • administer test data (database)
Designer		Identifies and defines the operations, attributes, and associations of the test classes. Responsibilities: • identifies and defines the test classes • identifies and defines the test packages
Implementer		Implements and unit tests the test classes and test packages. Responsibilities: • creates the test classes and packages implemented in the test model

4.2 System

[Which of our resources, could serve as targets-of-test?
Of what size are the test log files?
Is the test data amount a problem for our server hard disks?]

5 Project Milestones

5.1 Milestones Plan

[Feel free to identify milestones, e.g. testing hardware that must be ready by a time, test data that must be generated etc.

However, at least think of the following milestones:]

Milestone Task	Effort planned	Start Date planned	End Date planned
Plan Tests			
Design Tests			
Implement Tests			
Execute Tests			
Evaluate Tests			

[Explain how these milestones are assessed.]

5.2 Progress Assessment

[How are the testing status and effort updates reported to the management? Think of updating the table below at regular intervals. Use the deviation of the two effort columns to revise your time plan and get a more realistic effort estimations.

For status assessments and reports to the management you might use a table that shows the test progress as follows:

Test	total effort planned	lastest effort estimation	current effort	designed	implemented	reviewed	running error free
ST-XX-FT01	3 person days	5 days	4 days	100%	100%	yes	90%
ST-XX-FT02	4 days	5 days	4 days	100%	100%	no	0%
ST-XX-FT03	2 hours	2 hrs	0.5 hrs	100%	50%	no	0%
ST-XX-IT01	3 days	3 days	0 hrs	0%	0%	no	0%
...							

]

6 Deliverables

6.1 Test Model

This section identifies the reports that will be created and distributed.

6.2 Test Logs

[Describe the format (requirements) or methods and/or tools used to record and report on the test results.]

6.3 Defect Reports

[In this section identify the method and tools used to record, track, and report on test incidents and their status.]

7 References

7.1 Applicable Documents

In case of a conflict between a document listed below and this document, the listed document is applicable. However, such a conflict should be brought to the immediate attention of the project manager and document author.

[A1]	[Doc Number]	[Project and Document title]
[A2]	...	

7.2 Reference Documents

In case of a conflict between a document listed below and this document, this document is applicable. However, such a conflict should be brought to the immediate attention of the project manager and the author of the listed document.

[R1]	[Doc Number]	[Project and Document title]
[R2]	Rational	\\GNOME\RUP\RationalUnifiedProcess\index.htm

Advanced Coverage Metrics for Object-Oriented Software

Executive Summary

The use of structural coverage metrics to measure the thoroughness of a test set is a well-understood technique. However, the application of the technique to object-oriented software presents new challenges. This paper presents object-oriented context coverage, a new way to measure coverage for object-oriented systems.

IPL is an independent software house founded in 1979 and based in Bath. IPL has been accredited to ISO9001 since 1988, and TickIT accredited since 1991. IPL has developed and supplies the AdaTEST, Cantata and Cantata++ software verification products. AdaTEST, Cantata and Cantata++ have been produced to these standards.

Copyright

This document is the copyright of IPL Information Processing Ltd. It may not be copied or distributed in any form, in whole or in part, without the prior written consent of IPL.

IPL
Eveleigh House
Grove Street
Bath
BA1 5LR
UK
Phone: +44 (0) 1225 444888
Fax: +44 (0) 1225 444400
email ipl@iplbath.com

Certificate Number FM1589

Last Update:28/10/99 10:28
File: Advanced Coverage Metrics.doc

1. Introduction

The use of structural coverage metrics to measure the thoroughness of a test set is a well-understood technique. However, the application of the technique to object-oriented software presents new challenges.

Traditional structural coverage metrics such as statement coverage, branch coverage and condition coverage measure how well the bodies of each method have been tested. Unfortunately these traditional metrics do not take into account the object-oriented features of the software under test. In particular, the use of polymorphism and the encapsulation of state-dependent behaviour behind well-defined class interfaces are effectively ignored. Since polymorphism and encapsulation of behaviour are major features of any object-oriented design, metrics which ignore them are insufficient for determining whether the software under test has been thoroughly tested.

2. A New Way to Measure Coverage

To address this Cantata++ introduces an extension to traditional structural coverage – ***context coverage***. Context coverage is a way of gathering more data on how the software under test executes.

The context coverage approach can be applied to the testing of both polymorphic and state-dependent behaviour. The approach can also be extended to aid in the testing of multi-threaded applications.

By using these additional ***object-oriented context coverage*** metrics, in combination with traditional coverage metrics, we can ensure that the structure of the code has been fully exercised and thus have high confidence in the quality of our test set.

Three varieties of OO context coverage metrics are defined. ***Inheritance context coverage*** metrics are designed to help measure how well the polymorphic calls in the system have been tested. ***State-based context coverage*** metrics are designed to improve the testing of classes with state-dependent behaviour. ***User-defined context coverage*** metrics are also supported, allowing the application of the context coverage approach to other cases where traditional structural coverage metrics are inadequate, such as multi-threaded applications.

3. How Should We Test Polymorphism?

Traditional structural coverage metrics are inadequate as a measure of how well polymorphic[1] code has been tested. Consider the following fragment of code:

```
class Base {
public:
      void foo()                    { ... helper(); ... }
      void bar()                    { ... helper(); ... }
```

[1] Polymorphism is the ability to perform operations on an object without knowing exactly how the object will implement the operation. In C++, polymorphism can be achieved at run-time through the combination of several language features: (public) class inheritance, overriding virtual methods and using base class pointers or references to refer to derived class objects. Polymorphism can also be achieved at compile-time using templates.

©IPL Information Processing Ltd

```
private:
      virtual void helper()         { ... }
};
class Derived : public Base {
private:
      virtual void helper()         { ... }
};
void test_driver() {
      Base    base;
      Derived derived;
      base.foo();                   // Test Case A1
      derived.bar();                // Test Case A2
}
```

In this example, test case A1 invokes `Base::foo()` on the base object which in turn calls virtual function `helper()` on the base object, invoking `Base::helper()`.

In test case A2, note that `bar()` is not overridden in `Derived`, so the inherited method `Base::bar()` is invoked on the derived object, which in turn calls `helper()` on the derived object, invoking `Derived::helper()`.

Does the `test_driver()` function fully exercise the `Base` class? Does it fully exercise the `Derived` class?

Let's assume for now that the functions each contain linear code only – there are no decisions or loops at all.

Using a traditional structural coverage metric (such as statement coverage) as our guide we would answer yes to both questions since running `test_driver()` would achieve 100% coverage of `Base::foo()`, `Base::bar()`, `Base::helper()` and `Derived::helper()`.

3.1. Compile-Time Polymorphism With Templates

The C++ template feature provides a form of compile-time polymorphism. As with inheritance-based polymorphism, a routine from one class can be invoked on objects of different types. The template class corresponds to the base class where the routine is defined. The instantiations correspond to derived classes which inherit the routine.

The template routine can in turn call "helper" routines, where the particular helper routine invoked will depend on the type of the object. For example, a list template might invoke the contained object's copy constructor. The copy constructor invoked will be the one corresponding to the type of the contained object, which will be specific to a particular instantiation of the list template.

For the remainder of this paper, only inheritance-based (run-time) polymorphism will be discussed. However, the principles and techniques involved apply equally to template-based (compile-time) polymorphism.

3.2. What Did We Miss?

We have not fully tested the interactions between `Base` and `Derived`. Specifically, we have not tested the interactions between `Base::bar()` and `Base::helper()` or the interactions between `Base::foo()` and `Derived::helper()`.

©IPL Information Processing Ltd

Obviously, just because `Base::foo()` works with `Base::helper()` does not mean that it will automatically work when used with `Derived::helper()`. Although `Derived::helper()` has the same interface as `Base::helper()`, they have different implementations.

For our testing to be *really* thorough we should exercise both `foo()` and `bar()` for both the base class and the derived class – as we would if they were explicitly overridden in the derived class.

Figure 1 shows that the inherited methods are not fully covered by the original test cases. The diagram includes the "make-believe" versions of `foo()` and `bar()`(shown enclosed in {braces}) which represent their inheritance in `Derived`.

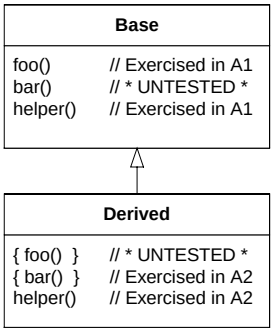

Base
foo() // Exercised in A1
bar() // * UNTESTED *
helper() // Exercised in A1
Derived
{ foo() } // * UNTESTED *
{ bar() } // Exercised in A2
helper() // Exercised in A2

Figure 1: The inherited methods have not been fully exercised

3.3. We Need Better Tests

To achieve 100% coverage an enhanced test set is required:

```
void better_test_driver() {
      Base    base;
      Derived derived;
      base.foo();                    // Test Case B1
      base.bar();                    // Test Case B2
      derived.foo();                 // Test Case B3
      derived.bar();                 // Test Case B4
}
```

The additional test cases help ensure that the interactions between the public methods and the private helper functions have been fully exercised, as shown in figure 2.

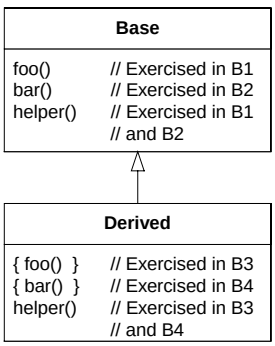

Base
foo() // Exercised in B1
bar() // Exercised in B2
helper() // Exercised in B1
// and B2
Derived
{ foo() } // Exercised in B3
{ bar() } // Exercised in B4
helper() // Exercised in B3
// and B4

Figure 2: Better test cases give 100% inheritance context coverage

©IPL Information Processing Ltd

3.4. Traditional Coverage Is Not Enough

It is clear that when methods are inherited by a derived class some further testing is necessary. This is true both for those methods which have been inherited unchanged and for those which have been overridden. Coverage measurement needs to be more context-dependent, recording the class of the specific object on which the method was executed (i.e. the context in which the coverage was achieved). Coverage achieved in the context of one derived class should not be taken as evidence that the method has been fully tested in the context of another derived class.

Unfortunately, traditional structural coverage metrics do just that. They treat any execution of a routine – whether it is in the context of the base class or a derived class – as equivalent. Thus `Base::bar()` is wrongly considered "100% covered" when it has been exercised in the context of class `Derived` only.

As figure 3 shows, OO code may be wrongly considered 100% covered by traditional metrics when in fact it has been 50% covered in the context of one derived class and 50% covered in the context of another derived class.

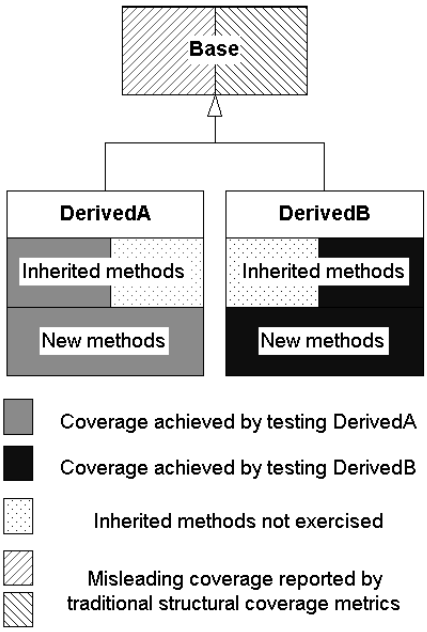


Figure 3: Only 50% coverage of inherited methods in each derived class yet Base appears fully tested using traditional metrics

This problem applies to *all* the traditional structural coverage metrics – none of them take into account the interactions between inherited base class methods and the overridden methods in each derived class.

3.5. What Is Inheritance Context Coverage?

Inheritance context coverage is not a single metric, but rather a way of extending the interpretation of (any of) the traditional structural coverage metrics to take into account the additional interactions which occur when methods are inherited.

Inheritance context coverage provides alternative metric definitions which consider the levels of coverage achieved in the context of each class as separate measurements. The inheritance context definitions regard execution of the routine in the context of

©IPL Information Processing Ltd

the base class as separate from execution of the routine in the context of a derived class. Similarly, they regard execution of the routine in the context of one derived class as separate from execution in the context of another derived class.

To achieve 100% inheritance context coverage, the code must be fully exercised in each appropriate context.

3.6. Definition

To take a specific example, the inheritance context coverage variant of decision coverage for a routine in a particular context is simply the number of decision branches exercised in the context divided by the total number of decision branches in the routine.

The overall ***inheritance context decision coverage*** for the routine is then defined as the average of the inheritance context decision coverage in each context appropriate to the routine. For a routine defined in a base class the appropriate contexts are the context which corresponds to the base class along with those corresponding to each derived class which inherits the routine unchanged. Note that the routine need not (more accurately, *can* not) be tested in the context of derived classes which provide an overriding definition of the routine.

Thus the overall inheritance context decision coverage for a routine is given by:

$$InheritanceContextDecnCoverage = \frac{\sum_{i=1}^{NumberOfContexts} DecnBranchesExercisedInContext_i}{NumberOfDecnBranchesInRoutine \times NumberOfContexts} \times 100\%$$

As with the standard coverage metrics, system-wide averages of inheritance context coverage (across all routines in the system) can also be defined.

3.7. An Example Using Decision Coverage

Let's assume that functions `foo()` and `bar()` shared significant amounts of common code. In an attempt to simplify the class, the functions are merged into one, with a boolean parameter determining the behaviour[2]:

```
class Base {
public:
      void foo_or_bar(bool flag) {
            ...
            if (flag) {
                  ...
                  helper();
                  ...
            } else {
                  ...
                  helper();
                  ...
            }
```

[2] This example is for the purposes of explanation only. In practice such a transformation would not normally be made – if "foo-ing" and "bar-ing" are separate enough concepts to have different names then it is unlikely that the merged method would be functionally cohesive. A better solution would be to extract the common code into separate private methods invoked by both `foo()` and `bar()`.

©IPL Information Processing Ltd

```
            ...
      }
private:
      virtual void helper()         { ... }
};
class Derived : public Base {
private:
      virtual void helper()         { ... }
};
void test_driver() {
      Base    base;
      Derived derived;
      base.foo_or_bar(false);       // Test Case A1
      derived.foo_or_bar(true);     // Test Case A1
}
```

The branches of `foo_or_bar()` have not been fully tested in the context of either `Base` or `Derived` – only 50% inheritance context decision coverage has been achieved in each case.

Note that in this example traditional decision coverage would misleadingly indicate 100% coverage of `Base::foo_or_bar()`.

3.8. What Are The Contexts?

In the examples above the valid contexts for the inherited methods (`Base::foo()` and `Base::bar()` or `Base::foo_or_bar()`) are simply "Base" and "Derived".

For the method `Base::helper()` the sole valid context is "Base", since the method is not inherited by `Derived` but is instead overridden.

For the overriding definition `Derived::helper()` the only valid context is "Derived".

3.9. Hierarchical Integration Testing

Let us consider the level of testing required for inherited methods which are inherited, using the example shown in figure 4.

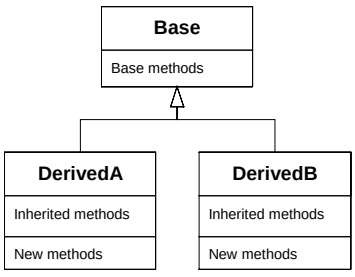


Figure 4: The software under test

The Hierarchical Integration Testing (HIT) approach to unit testing was proposed as a technique for ensuring thorough class testing (see [Harrold] for full details). HIT recommends that as a first step all methods be tested fully in the context of a particular class (the base class or, for abstract base classes, a particular derived class).

For a base class this would typically be interpreted as a coverage requirement of 100% decision coverage of each defined method in the context of the particular class. This

©IPL Information Processing Ltd

criterion is also known as “Once-Full” coverage and is equivalent to the minimum traditional coverage requirement for thorough unit testing.

This recommendation applies to all classes, so that re-definitions of a method in a derived class (“overridden” methods) are tested with the same thoroughness as the original base class definition.

3.9.1. Interaction Coverage

The HIT approach further recommends that any methods which are inherited by a derived class and which interact with any re-defined methods should be re-tested in the context of the derived class. The focus of this re-testing is to exercise the interactions between the inherited methods and the re-defined method(s) – primarily the calls between the methods. This recommendation could be interpreted as a coverage requirement of 100% call-pair coverage of the inherited methods in the context of each derived class.

3.9.2. Strict Coverage

In practice the integration test cases which exercise the interactions between methods are often spread throughout the complete test suite for the class.

Rather than separating the integration test cases out, sometimes the easiest way to ensure that interactions are thoroughly re-tested is to re-run *all* of the base class test cases. This conservative approach can be enforced through the application of a stricter coverage requirement, namely that 100% decision coverage is achieved for each base class method in the context of *every* derived class by which it is inherited.

3.10. Achieving Inheritance Coverage Is Easy

During unit testing the effort required to achieve inheritance context coverage is not significantly greater than that required to achieve coverage according to traditional metrics.

Typically, no additional test cases are required. Instead, test cases already written to test the base class are used to re-test the inherited methods in the context of the derived class. The test cases form a ***parallel inheritance hierarchy*** which mirrors the inheritance structure of the software under test and enables base class test cases to be easily re-used to test derived classes (see Figure 5).

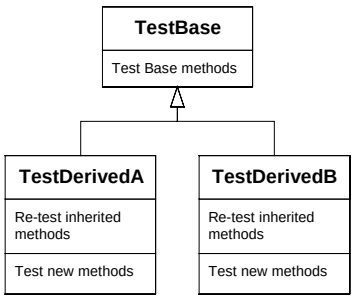


Figure 5: Test classes and software under test form parallel inheritance hierarchies

©IPL Information Processing Ltd

This re-use of base class test cases has the additional benefit of automatically testing the design for conformance to the Liskov Substitutability Principle (LSP). The LSP is an important object-oriented design principle (described in [Liskov]) which helps ensure that inheritance hierarchies are well-defined. The use of a parallel inheritance hierarchy also forms the basis of the PACT method described in [McGregor].

The only costs for this re-use are the (small) initial effort to ensure that the test cases are structured so as to be re-usable and the CPU time required to re-run the test cases.

4. State-Based Context Coverage

In most object-oriented systems there will exist a number of classes which can best be described as "state machines". Objects of these classes can exist in any of a number of distinct states, and the behaviour of each class is qualitatively different in each possible state – the behaviour of the class is state-dependant. ***State-based context coverage*** is designed to measure how fully this behaviour has been tested.

Consider a typical class with state-dependent behaviour: a bounded stack. A bounded stack can be in one of three possible states: 'empty', 'partially full' or 'full'. The behaviour of the stack is qualitatively different in each state. For example, the `pop()` operation removes and returns the top element from the stack in states 'partially full' or 'full' but throws an exception and leaves the stack unmodified in state 'empty'.

The class interface for the bounded stack class is shown below:

```
class BoundedStack {
public:
      BoundedStack(size_t maxsize);
      ~BoundedStack();
      void push(int);
      int pop();
      struct underflow : std::exception { };
      struct overflow : std::exception { };
};
```

4.1. Black-Box Testing using Entry-Point Coverage

When testing a class, the first step is to write test cases to exercise the public interface to the class:

```
int test_driver() {
      BoundedStack stack(2);
      stack.push(1);
      stack.pop();
      // destructor called implicitly at end of block
};
```

We can use entry-point coverage to ensure that the test cases exercise each of the methods of the class. The test cases above achieve 100% entry-point coverage.

However, it is clear that this test set does not fully exercise the `BoundedStack` class; the stack never becomes full and an exception is never thrown.

©IPL Information Processing Ltd

4.2. Using White-Box Coverage Metrics

If entry-point coverage does not ensure thorough testing, perhaps we should use a stronger coverage requirement, say, 100% decision coverage. Unfortunately, there are disadvantages to the adoption of such a requirement.

One factor is that there may be decisions in the code which do not correspond to features of the public interface. Typical examples include error handling code and defensive programming idioms. In these cases, 100% decision coverage may be difficult to achieve.

A more significant problem is the inability of traditional structural coverage metrics to identify code which is missing altogether. For example, consider what would happen if the `BoundedStack` implementor forgot to check for the stack-empty condition in `pop()`. Requiring 100% decision coverage would not help find this fault – the missing condition is not there to be covered!

4.3. We Can Do Better

Actually, we can write a better test set than that required by any traditional structural coverage metric, and without any knowledge of the internal details of the class.

The UML state transition diagram (see figure 6) shows how the behaviour of the class changes, depending on the current state. This type of diagram is commonly used to describe the state-dependent behaviour of classes

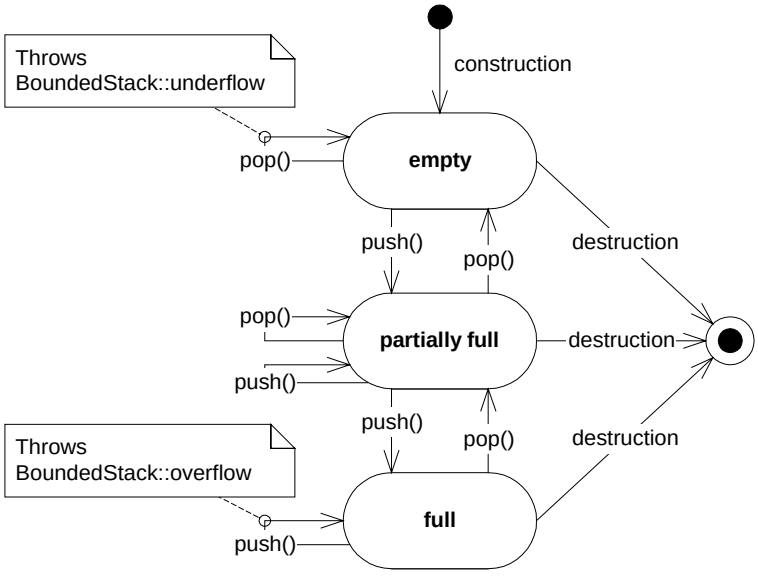


Figure 6: State transition diagram for BoundedStack

We can use the additional information provided in the state transition diagram to design a test set which thoroughly exercises the `BoundedStack` class.

4.4. Writing State-Driven Test Cases

The aim of our test design is to exercise every method in every possible state. In the improved test set which follows, the `push()` and `pop()` methods are invoked on an empty, partially full and full stack:

©IPL Information Processing Ltd

```
int better_test_driver() {
      BoundedStack stack(2);
      stack.push(3);                // push() when empty
      stack.push(1);                // push() when partially-full
      try { stack.push(9); }        // push() when full
      catch (BoundedStack::overflow) { }  // expected to throw
      stack.pop();                  // pop() when full
      stack.pop();                  // pop() when partially-full
      try { stack.pop(); }          // pop() when empty
      catch (BoundedStack::underflow) { } // expected to throw
      // destructor called implicitly at end of block
};
```

The crucial issue is: does this test set fully exercise the `BoundedStack` class?

State-based context coverage is designed to answer this question.

4.5. State-Based Context Coverage

State-based context coverage is similar to inheritance context coverage: it provides alternative definitions of the traditional structural coverage metrics. The alternative definitions differ in that they separately measure coverage in different contexts.

With inheritance context coverage the contexts depend on the inheritance structure of the software under test. Similarly, with state-based context coverage the contexts correspond to the potential states of objects of the class under test.

Thus the state-based context coverage metrics regard execution of a routine in the context of one state (that is, when invoked on an object in that state) as separate from execution of the same routine in another state. To achieve 100% state-based context coverage, the routine must be exercised in each appropriate context (state).

4.6. Definition

The state-based context coverage variant of entry-point coverage for a class (in the context corresponding to a particular state) is given by the number of methods which have been invoked on objects in that state divided by the total number of methods in the class.

The overall state-based context entry-point coverage is then defined as the average of the state-based context entry-point coverage in each state appropriate to the class.

Thus the overall state-based context entry-point coverage for a routine is given by:

$$StateBasedContextEntryPointCoverage = \frac{\sum_{i=1}^{NumberOfStates} MethodsExercisedInState_i}{NumberOfMethodsInClass \times NumberOfStates} \times 100\%$$

Note that, for the purposes of state-based context coverage, constructors are not considered to be methods of the class, in the sense that when they are invoked the object does not (yet) exist, and thus there is no state with which to associate the coverage.

Alternatives are similarly defined for each of the traditional structural coverage metrics.

©IPL Information Processing Ltd

As with the standard coverage metrics, system-wide averages of state-based context coverage (across all classes in the system) can also be defined.

4.7. What Are The Contexts?

The contexts for state-based context coverage are just the states appropriate to the class. Thus, in the bounded stack example the contexts are “empty”, “partially-full” and “full”.

Unfortunately, the presence of these states is a design feature which is not directly visible in the code. In order for a tool to automatically gather state-specific coverage information, the code must be changed so that it defines the possible states and so that the tool can determine the current state.

The simplest solution is to define an additional member function which returns a string containing the current state. For example:

```
class BoundedStack {
public:
      ...
private:
      const char* cppca_user_context_function() const {
            if (num_elements == 0) return "empty";
            else if (num_elements == max_elements) return "full";
            else return "partially-full";
      }
};
```

When provided by the user, this specially-named member function is used automatically by Cantata++ to determine the current state while gathering coverage data.

4.8. How Well Did We Do?

Using the test cases from `better_test_driver()` to test the `BoundedStack`, we find that we still haven’t achieved 100% state-based entry-point coverage. The destructor has not been exercised in the “partially-full” or “full” states.

Although state-based coverage does not apply to constructor functions, it does apply to destructors. Exercising destructors in all states helps ensure that all allocated resources are properly freed.

The following, further enhanced, test set does achieve 100% state-based entry-point coverage:

```
int even_better_test_driver() {
      BoundedStack stack(2);
      stack.push(3);                // push() when empty
      stack.push(1);                // push() when partially-full
      try { stack.push(9); }        // push() when full
      catch (BoundedStack::overflow) { }  // expected to throw
      stack.pop();                  // pop() when full
      stack.pop();                  // pop() when partially-full
      try { stack.pop(); }          // pop() when empty
      catch (BoundedStack::underflow) { } // expected to throw
      BoundedStack stack2(3);
      stack2.push(6);               // stack2 is partially-full
      BoundedStack stack3(1);
```


©IPL Information Processing Ltd

```
      stack3.push(6);               // stack3 is full
      // destructors called implicitly at end of block for
      // stack (empty), stack2 (partially-full) and stack3 (full)
};
```

4.9. Extra Effort Gives Extra Benefit

State-based context coverage is unusual because it requires the user to provide additional information (in the form of the additional member function to return the current state).

In contrast, traditional structural coverage metrics such as decision and condition coverage are based purely on the structure of the software under test and therefore require no additional information. As a result, traditional structure coverage metrics are typically weak at highlighting "faults of omission" – requirements which have simply not been implemented. Suppose, through an oversight, that the code to implement a particular requirement is missing. There is no hint in the code that the requirement exists, and no traditional structural coverage metric will force the testing of the missing requirement.

The use of an independent source of additional information means that state-based context coverage has the potential to go beyond the constraints of traditional structural coverage metrics and highlight these faults of omission.

For example, areas of unimplemented functionality can be identified by analysing those areas for which state-based entry-point context coverage has not been achieved while 100% traditional (non-context) decision coverage has been achieved.

4.10. Achieving State-Based Coverage

Achieving 100% state-based entry-point coverage typically requires more test cases than are required for traditional entry-point coverage and fewer than are required for traditional decision coverage. Moreover, because the test cases which are required to achieve state-based entry-point coverage are based on the design they are usually just those which would form a normal black-box test set, particularly if an approach such as [Binder]'s "FREE" is used. This reduces the need to write additional test cases purely to achieve a defined coverage target.

5. Thread-Based Context Coverage

The concept of context coverage which underpins both inheritance context coverage and state-based context coverage can also be used to assist with other coverage analysis problems.

For example, when testing a multithreaded application, traditional structural coverage metrics will "aggregate" the coverage achieved by all the threads into a single coverage value.

The context coverage approach can be applied here to maintain separate coverage information for each thread. As with state-based context coverage, a user-provided function is used automatically by Cantata++ to determine the current context:

©IPL Information Processing Ltd

```
const char* cppca_user_context_function() const {
      static char buffer[16];
      sprintf(buffer, "%x", OS::Threads::GetCurrentThreadID());
      return buffer;
}
```

The additional coverage information provided through context coverage can be used o ensure that each individual thread is thoroughly tested. Moreover, the raw coverage data (which statements and decisions are executed in which threads) can be used to analyse how the software under test actually behaves in the presence of complex thread-interaction issues.

6. Conclusion

Traditional structural coverage metrics are inadequate measures of test thoroughness for object-oriented software systems. New object-oriented context coverage metrics are required to ensure thorough testing. Inheritance, state-based and thread-based context coverage metrics are a useful and practical addition to the unit tester's tool set.

Inheritance context coverage can be used, in conjunction with traditional structural coverage metrics and with little additional cost to ensure that polymorphic interactions between methods are fully tested in each derived class. Similar techniques can be applied to uses of template-based compile-time polymorphism.

State-based context coverage can be used, again in conjunction with traditional structural coverage metrics, to ensure that classes whose behaviour depends on an internal state have been thoroughly tested. In particular, state-based entry-point coverage is an appropriate metric for black-box unit-testing of classes with state-dependent behaviour.

The underlying techniques of context coverage can also be extended to apply to assist with other coverage problems, such as the testing of multi-threaded applications.

7. References

[Binder] Binder,B., *The FREE Approach to Testing Object-Oriented Software: An Overview*, http://www.rbsc.com/pages/FREE.html.

[Harrold] Harrold,M.J. and J.D.McGregor, *Incremental Testing of Object-Oriented Class Structures*, http://www.cs.clemson.edu/~johnmc/papers/TESTING/HIT/hit.ps.

[Liskov] Liskov, B. and J.Wing, *A Behavioral Notion of Subtyping*, ACM Transactions on Programming Languages and Systems, Vol 16, No 6, November, 1994, pages 1811-1841.

[McGregor] McGregor,J.D. and A.Kare, *PACT: An Architecture for Object-Oriented Component Testing*, Proceedings of the Ninth International Software Quality Week, May 1996.

©IPL Information Processing Ltd

Codierungsstandards in der Software-Entwicklung für Eingebettete Systeme

Codierungsstandards haben in die Software-Entwicklungsabteilungen von Unternehmen verschiedenster Branchen Einzug gehalten. Diese oft firmenspezifischen Standards schreiben den vor in welchem Stil ein Software-Ingenieur zu programmieren hat, wobei mächtige Standards nahezu alle Aspekte der Programmierung abdecken, vom Schriftbild des Programms bis zu Designrichtlinien oder -einschränkungen. Der folgende Artikel beleuchtet die Bedeutung solcher Standards speziell für den Bereich Eingebettete Systeme und gibt einen Überblick über die Themen die ein Codierungsstandard behandeln sollte. Auch Probleme, die mit der Einführung eines solchen Standards in einen Betrieb einhergehen werden behandelt.

Sprachstandards nicht nur für Technik & Software

Einer der vielen Nutzen von technischen Standards ist das Ausschalten von leicht vermeidbaren Fehlern und das messbare Einhalten von Qualitätskriterien. Das Lösen von Problemen in einer standardisierten Art und Weise macht es leichter die Lösung zu verstehen und vermeidet unnötige Neuentwicklungen. So kann auch die Einführung von Standards in einer natürlichen Sprache Missverständnisse ausräumen und die Eindeutigkeit von Aussagen erhöhen. Ein gutes Beispiel dafür ist die Verwendung der Juristensprache bei der Erstellung von Verträgen. Die dort verwendeten Redewendungen sind im Idealfall unmissverständlich und exakt. Als solche genügen sie den Anforderungen eines Vertragstexts und schützen die Unterzeichner vor Unannehmlichkeiten oder Gesetzeswidrigkeit des Vertrags. Allerdings ist eine solche Fachsprache nicht gerade für jedermann verständlich. Sie muss erlernt werden. Ist sie jedoch einmal erlernt, so ist das schnelle und unmissverständliche Formulieren und Verstehen komplexer juristischer Sachverhalte wesentlich leichter als zuvor.

Formale Sprachen, also Programmiersprachen, sind von sich aus unmissverständlich, weil sie sich sonst nicht in Maschinenanweisungen übersetzen lieβen. Allerdings oft nur für den Computer, denn jeder, der einmal ein Computerprogramm fragwürdigen Stils studieren musste, weiβ wie leicht Fehlinterpretationen zustandekommen und wie schwer es oft ist dem Programm zu folgen. Genau wie in der natürlichen Sprache helfen hier Standards. Anders als in der natürlichen Sprache muss dieser Standard aber nicht erst erlernt werden um dem Leser die Interpretation von Quelldateien zu erleichtern. Die Kenntnis des Standards erleichtert allerdings die Interpretation und verkürzt damit gegebenenfalls die Einarbeitungszeit eines neuen Mitarbeiters.

Warum Codierungsstandards?

Eine verkürzte Einarbeitungszeit neuer Mitarbeiter ist bei weitem nicht der einzige Grund der die Beachtung eines Codierungsstandards besonders im Bereich Eingebettete Systeme rechtfertigt. Die Anwendung eines guten Codierungsstandards

- erleichtert das Schreiben von portierbarer Software
- erleichtert die Erstellung von robusten und zuverlässigen Programmen
- kann den Testaufwand erheblich reduzieren
- erhöht die Wartbarkeit des Codes
- erhöht die Lesbarkeit des Codes und erleichtert damit die Kooperation von Programmierern

Dies gilt jedoch nur wenn der Einsatz eines Codierungsstandards *rechtzeitig* erfolgt. Von der gewaltsamen Einführung von Codierungsstandards in laufenden Projekten, in denen bereits Codiert wird, ist eher abzuraten. Ein Codierungsstandard ist nur rentabel, wenn er im gesamten Projektzeitraum Anwendung findet!

Im folgenden widmen wir uns den Themen, die ein solcher Standard abdecken sollte (oder abdecken kann). Er umfasst meist allgemeingültige Vorschriften, die unabhängig von der verwendeten Programmiersprache sind, und sprachabhängige Bestimmungen. Wir folgen diesem Muster und wenden uns zunächst allgemeingültigen Vorschriften zu. Im Normalfall unterscheidet ein Standard zwischen Regeln, die beachtet werden *müssen*, und Empfehlungen, die beachtet werden *sollten*.

Lexikalische Standardisierung

Lexikalische Regeln betreffen etwa die Verwendung von Abkürzungen in Variablennamen, die Verwendung von Nummern und Zahlenformaten, Groβ- und Kleinschreibung sowie reservierte Namen. Viele Standards haben auch sehr detaillierte Vorschriften zur Benennung von Datentypen und Variablen. Diese Vorschriften umfassen meist die Namensgebung selbst und eine formal spezifizierte Vorsilbe (Präfix) bzw. Nachsilbe (Postfix). Ein Beispiel hierfür ist die von Charles Simonyi erfundene, oft zitierte und mutierte Ungarische Notation [1].

Auch Dateinamen und -platzierungen werden gelegentlich standardisiert. CASE-Tools machen derartige Regeln meist überflüssig. Jedoch ist der Einsatz von derartigen Werkzeugen für eingebettete Software oft nicht machbar oder in vielen Fällen nicht rentabel. Daher ist die Beachtung dieses Themas im Standard empfehlenswert.

Vorschriften an das Code Layout

Nahezu alle gängigen Codierungsstandards weisen eine Fülle von Regeln auf, die das Layout des Programms betreffen. Einheitliche Programmstrukturen erhöhen nicht nur die Lesbarkeit sondern auch die Wartbarkeit von Programmen. Eingebettete Software wird selten mit mächtigen Entwicklungsumgebungen erzeugt, die es erlauben sofort alle aufrufenden oder aufgerufenen Unterprogramme eines Programmblocks anzuzeigen. Oft steht nur ein Compiler für die Kommandozeile und ein einfacher Editor zur Verfügung. Ohne genaue Vorschriften an die Klammernsetzung kann dann etwa das Suchen von Funktionsaufrufen in umfangreichem C-Code zum Alptraum werden.

Layout-Vorschriften umfassen neben der Klammernsetzung meist Regeln zur Verwendung des Leerzeichens, zum Einsatz von Leerzeilen und Blockstrukturen, Vorschriften zur Einrückung sowie zur vertikalen und horizontalen Ausrichtung von Text.

Kommentare

Kommentare bereichern das Programm mit Zusatzinformation für den Leser und sollen helfen die Interpretation des Codes zu erleichtern. Es ist überflüssig zu erwähnen, dass falsche Kommentare noch schlechter sind als keine Kommentare. Die Verständlichkeit von Kommentaren kann wohl schlecht standardisiert werden, daher betreffen Regeln für Kommentare meist was kommentiert werden muss und welche jeweiligen Vorlagen zu verwenden sind. Etwa Vorlagen für das Markieren von Unterprogrammen oder Sprungadressen. Standards für Kommentare sind also zu einem guten Teil Anweisungen an das Layout der Kommentare.

Oft sind die Erwähnung des Autors, des Projekts, Erstellungsdatum und derengleichen standardisiert und im Idealfall mit einem System zur Versionskontrolle abgestimmt.

Sprachunabhängige Anforderungen an das Design

Häufig findet man in Codierungsstandards auch Kapitel mit Anforderungen, die das Softwaredesign betreffen und damit nicht mehr dem reinen Codieren zugeschrieben werden können. Diese sind in der Regel schwer von der gewählten Programmiersprache zu trennen, genau wie das Design selbst. Viele Standards verzichten daher auf starre Regeln und es wird mehr auf Ecksteine des sauberen Softwaredesigns hingewiesen. Der Programmierer wird meist angehalten sich an Prinzipien wie Kohäsion, Modularität, Abstraktion und derengleichen zu halten..

Für höhere Programmiersprachen, also alles was über Assembler hinausgeht, werden in Standards gelegentlich Softwaremetriken definiert, die eingehalten werden müssen. Derartige Metriken definieren etwa die maximale Größe von Unterprogrammen, die maximale Verschachtelungstiefe von Schleifen oder Verzweigungen pro Unterprogramm, oder etwa einen Mindestprozentsatz an Kommentarzeilen. Diese sogenannten *statischen Softwaremetriken* werden mit Hilfe von Hilfsprogrammen errechnet. Die Überprüfung, ob die erstellte Software den Metriken genügt, ist dann entweder Teil einer Code-Review oder ein Aspekt des Modultests.

Die Metriken werden „statisch" genannt, weil zu ihrer Berechnung die Software nicht ausgeführt werden muss, sondern nur der syntaktisch richtige Code genügt um sie zu berechnen. Dynamische Softwaremetriken hingegen beschreiben die Effizienz von Tests und nicht die Software selbst.

Sprachabhängige Anforderungen an das Design

Wenn der Codierungsstandard *eine konkrete* Programmiersprache betrifft, dann ist das Formulieren von Anforderungen an das Design schon wesentlich leichter. Mehr als alle bisher vorgestellten Regeln sollen die sprachspezifischen Designvorschriften vermeiden in bekannte Fallen einer Programmiersprache zu tappen. Dazu selbst braucht man zwar nicht unbedingt einen Codierungsstandard, weil es gute Literatur am Büchermarkt [2], Beiträge im WWW [3], in Zeitschriften [4] und im günstigsten Fall automatische Erkennungssysteme [5] gibt, doch sollten derartige Vorschriften im Standard nicht fehlen.

Im einfachsten Fall bestehen die sprachabhängigen Designanforderungen aus Verboten von Sprachkonstrukten. So verbieten die meisten Codierungsstandards für C die Verwendung der `goto`-Anweisung und einige C++ Standards die Verwendung von Objektattributen der Klassifikation `protected`. Neben der Beachtung von Verboten kann auch Befolgung von Geboten die Robustheit und Zuverlässigkeit des Programms erhöhen. Beispielsweise ist bei der Verwendung von Mehrfachvererbung oder beim Überladen von Operatoren Vorsicht geboten. Diese Themen sind daher Gegenstand von vielen Codierungsstandards für C++.

Ein guter Codierungsstandard widmet sich auch ausführlich dem Thema Fehlerbehandlung. Vor allem in C ist dies schon historisch gesehen ein Problem für sich selbst. Nicht einmal die Erfinder der Programmiersprache konnten sich auf ein einheitliches Design bei der Fehlerbehandlung einigen.

Regeln für das Übersetzen

In Codierungsstandards wird häufig auf Richtlinien zur Übersetzung des Programms vergessen. Die Programmierung ist mit dem Erstellen des Quelltextes noch nicht vorüber. Je nach verwendeter Sprache sollte man Richtlinien für Compiler-Warnungen erstellen und Schalter für Sprachstandards empfehlen. Im günstigsten Fall gibt es eine Norm für Makefiles und die Verwendung einer Make-Utility.

Probleme bei der Einführung eines Codierungsstandards in ein Unternehmen

Wenn Sie heute in Ihrer Softwareabteilung erstmals einen Codierungsstandard einführen, so dürfen Sie nicht unbedingt mit Freudensprüngen Ihrer Mitarbeiter rechnen. Manche Softwareentwickler fühlen sich durch solche Standards in ihrer Kreativität eingeschränkt. Auch ist es schwer einen einheitlichen Programmierstil zu finden, der von allen (freiwillig) akzeptiert wird. Einige könnten ihren Programmierstil als eine persönliche Note sehen und sich daher nur ungern auf einen Standard einlassen.

Die Einführung eines Codierungsstandards verfehlt seinen Zweck, wenn

- der Standard von den Mitarbeitern nicht als wichtig anerkannt wird und daher
- er nicht rigoros eingehalten wird oder die Einhaltung nie überprüft wird
- die Software nur von einer einzigen Person entwickelt und gewartet wird und diese sehr erfahren und gewissenhaft ist

Um einer Ablehnung durch Mitarbeiter entgegenzusteuern, sollten diese vor der Einführung umfassend informiert werden und am besten am Standard mitarbeiten. Es ist auch wichtig anzumerken, dass die Einführung eines Codierungsstandards natürlich nicht automatisch gute Software garantiert und dies sollte auch bei weitem nicht die einzige Maßnahme zur Qualitätsverbesserung in der Softwareentwicklung sein. Ein Codierungsstandard ist aber ein Wegbereiter für höhere Qualität, bessere Wartbarkeit und schnellere Einarbeitungszeiten und kann somit Geld sparen, sobald in einem Softwareprojekt mehr als eine Personen arbeitet.

Zusammenfassung

Dieser Artikel informiert über die Motivation der Einführung von Software Coding Standards und bietet einen umfassenden Überblick über die Themen die in einem derartigen Standard bedacht werden sollten. Die Behandlung der Themen ist so weit gefasst, dass sie für alle relevanten Programmiersprachen im Bereich Eingebettete Systeme gültig ist.

Referenzen

[1] http://msdn.microsoft.com/library/techart/hunganotat.htm
[2] Andrew Koenig: *C Traps and Pitfalls*, Addison-Wesley 1988.
[3] http://www.andromeda.com/people/ddyer/topten.html
[4] Angelika Langer: „Stolpersteinen aus dem Weg gehen“, *Elektronik* 6/1998.
[5] Software wie *PC-lint* oder *lint*. Siehe zum Beispiel http://www.gimpel.com

Ein guter Programmstil will standardisiert sein

Quellcode eines erfahrenen Programmierers unterscheidet sich meist schon im Erscheinungsbild von Quellcode, der von einem Anfängern erstellt wurde. Profis halten sich konsequent an einen durchdachten Stil. Wenn so ein Programmierstil in Form von Regeln und Empfehlungen dokumentiert ist, dann können sich Anfänger diesen leichter aneignen und selbst schneller Profis werden. Ist die Einhaltung dieser Regeln für alle Mitarbeiter eines Projekts verpflichtend, so sprechen wir von einem Software-Coding-Standard. Der folgende Artikel präsentiert Regeln zur Verwendung von Zahlen und Namensgebung die auf eine Vielzahl von Programmiersprachen anwendbar sind. Deren Beachtung erhöht die Lesbarkeit des Quellcodes, die Zusammenarbeit im Projekt und die Wartung der Software wird damit vereinfacht.

Elementarteilchen des Quellcodes

Bei der Erstellung der erwähnten Regeln müssen wir dort beginnen, wo das „Leben" eines Programms beginnt: bei der Verwendung von Buchstaben, Ziffern und der natürlichen Sprache. Keine Angst, Esoterik hat hier nichts verloren. Vielmehr definieren wir zunächst Bausteine, die wir dann zu mächtigen Konstrukten zusammenfügen werden.

Die erste Frage, die Sie sich stellen müssen, ist die, welche natürliche Sprache Sie zur Dokumentation verwenden. In vielen Fällen wird dies Englisch sein, folglich muss dann auch Ihr Coding-Standard auf Englisch sein. In unserem Artikel werden wir uns jedoch weiter an die deutsche Sprache halten. Damit die Inhalte des Artikels besser in einen englisch verfassten Standard übernommen werden können, verwenden wir englischsprachige Bezeichner und formulieren daher unsere erste

Regel 1: Dokumentationssprache

> Alle Software-Dokumente und Programmkommentare müssen in deutscher Sprache geschrieben sein, benutzerdefinierbare Bezeichner in Programmen müssen englisch sein.

Nahezu jeder Quereinsteiger stolpert über verwendete projekt- oder firmenspezifische Abkürzungen. Die Führung eines Abkürzungsverzeichnisses ist daher unabdingbar.

Regel 2: Abkürzungen

> Alle verwendeten projektspezifischen Abkürzungen müssen im Designdokument erklärt werden. Projektübergreifende Abkürzungen werden im Anhang des Coding-Standards definiert.

Alternative Formulierungen dieser Regel können auf elektronische Abkürzungsverzeichnisse verweisen. Ein solches Verzeichnis bietet sich insbesondere für projektübergreifende Kurzformen an. Beispielsweise unter Verwendung der Kommandos `man` und `whatis` in Unix. Ebenso wichtig kann auch ein Querverweis auf eine weitere Erläuterung sein. Folgendes Beispiel aus einem Abkürzungsverzeichnis ist nicht erfunden, sondern stammt aus einem Dokument einer großen deutschen Elektronikfirma:

> Abkürzung: DSS Bedeutung: Datensammelschiene

Dies zeigt, dass ein schlecht gewählter Begriff genauso nichtssagend sein kann wie seine Abkürzung. Daher formulieren wir folgende

Regel 3: Projekt Glossar

> Alle projektspezifischen Definitionen oder Abkürzungen müssen in einem Glossar erklärt werden. Das Designdokument muss zumindest eine Referenz auf diesen Glossar enthalten.

Nun wenden wir uns den Zahlenformaten zu. Die Verwendung von Oktalzahlen hat sich nie wirklich durchgesetzt. Einer der Gründe ist, dass sie zu leicht mit Dezimalzahlen verwechselt werden können. Daher folgende

Empfehlung 4: Verwendung von Oktalzahlen

> Oktalzahlen sollten nicht verwendet werden.

Der Grund für die Verwendung dieses Zahlenformats und dessen Unterstützung von einigen aktuellen Programmiersprachen ist die einfache Umrechnung zwischen Oktal- und Binärzahlen. Den gleichen Vorteil haben die deutlich häufiger verwendeten Hexadezimalzahlen. Sie werden dann verwendet, wenn das Bitmuster einer Zahl eine besondere Rolle spielt.

Empfehlung 5: Zahlenformat

> Das verwendete Zahlenformat sollte dem Umfeld der Verwendung der Zahl angepasst sein. Binärzahlen oder Hexadezimalzahlen sollten nur dann verwendet werden, wenn auf Hardware Rücksicht genommen werden muss, andernfalls sind Dezimalzahlen zu verwenden.

Regel 6: Buchstaben und Vorzeichen in Zahlenformaten

> Buchstaben in Hexadezimalzahlen müssen Großbuchstaben sein. Der Exponent bei Exponentialdarstellungen muss von der Mantisse durch ein kleines „e“ und ein Vorzeichen getrennt sein.

Die oben formulierte Regel ist eine willkürliche Festlegung auf ein einheitliches Format. Genauso gut könnten für Hexadezimalzahlen die Verwendung von Kleinbuchstaben vorgeschrieben werden oder ein großes „E“ für Exponentialdarstellung verlangt werden. Jedoch können bei Kleinbuchstaben die Ziffer 6 und der Buchstabe b leicht verwechselt werden, wenn etwa während der Fehlersuche Notizen gemacht werden. Das kleine „e“ in der Darstellung von Exponentialzahlen ist die Standardeinstellung von vielen Mathematikpaketen und von ANSI C++. Das Vorzeichen unterstreicht das Exponentialformat zusätzlich, vergleichen Sie die Literale `1.2345E6` und `1.2345e+06`!

Wenn wir Hexadezimalzahlen für Zugriffe auf die Hardware verwenden, ist es vorteilhaft, die vorhandene Wortbreite in der Zahlendarstellung zu reflektieren. Daher die folgende

Empfehlung 7: Ziffernzahl für Hexadezimalzahlen

> Die Anzahl der Stellen einer Hexadezimalzahl sollte die Anzahl der tatsächlich verwendeten Bits widerspiegeln.

Dazu Anwendungsbeispiele in C:

```
#define BITMASK 0x00000FF0     /* Maske für 32-bit-Werte      */
char cBlinkPattern = 0x0F;     /* LED = 400ms ein, 400 ms aus */
unsigned uiAdcZeroLevel = 0xF000;     /* 16-bit Bus-Interface */
```

Irreführend ist, wenn für eine 16-bit-Größe eine Hexadezimalzahl verwendet wird, deren Stellenanzahl auf ein 32-bit-Format schließen lässt. Um derartigen Unfug zu verbieten, können wir entweder die letzte Empfehlung in eine Regel umwandeln oder eine weitere entsprechene Regel formulieren, die das explizit verbietet.

Beides wird uns nicht davor schützen, dass Programmiersprachen prozessorabhängige Komponenten besitzen können und in C etwa der Typ `int` (abgeleitet von der internen Verarbeitungsbreite des Prozessors) z.B. 16 oder 32 bit breit sein kann. Hier kann die projektübergreifende Definition von Typen mit fester Stellenanzahl helfen. Beispielsweise verwenden manche Entwicklungswerkzeuge den Typ `INT32` für ganzzahlige 32-bit-Werte oder `UINT16` als 16-bit-Typ für positive ganze Zahlen in Form eines Makros. Für jeden Compiler bzw. Prozessor muss dann nur die entsprechende Typendefinition in einem Makro adaptiert werden, der Rest des Quellcodes ist jedoch von der Adaption nicht betroffen und somit leichter portierbar. Ihr persönlicher Codierungs-Standard kann eine entsprechende Regel vorsehen, die die Verwendung dieser Typen vorschreibt. Da eine derartige Regel sprachabhängig ist und wir uns in diesem Artikel mit weitgehend sprachunabhängigen Aspekten befassen, wird auf dieses Thema nicht weiter eingegangen.

Motivation einer guten Namensgebung

Eine der wichtigsten Stützen eines sauberen Programmierstils ist die Wahl aussagekräftiger Bezeichner für Module (Quelldateien), Funktionen, Datentypen und Variablen. In diesem Artikel werden englischsprachige Namen verwendet, weil nahezu alle kommerziellen Bibliotheken und die Programmiersprachen selbst Bezeichner in dieser Sprachen haben.

Ein häufiger Fehler von unerfahrenen Programmierern ist die Verwendung von zu kurzen bedeutungslosen Namen oder die unsachgemäße Verwendung von Fachbegriffen. Hier kann mit einem Coding-Standard entgegengesteuert werden. Er sollte eine Liste von Fachbegriffen beinhalten, deren Bedeutung erläutern und die Verwendung dieser Begriffe in einem anderen Kontext verbieten. Um die Wichtigkeit dieser Standardisierung zu unterstreichen, hier ein paar Programmzeilen, die die Regeln der guten Namensgebung schmerzlich verletzen:

```
a = GetStack();       /* hole oberstes Element vom Stack      */
l = GetLatchValue();  /* l = neuer Wert für das Register      */
LatchValue(l);        /* schreibe jetzt den Wert ins Register */
WriteStack(l);        /* und sichere ihn auf dem Stack        */
```

Die Wahl der Funktionsnamen ist mehr als nur schlecht. Das Präfix `Get` wird im allgemeinen verwendet, wenn das Object der Operation dabei unverändert bleibt. Der Kommentar beschreibt aber eine Operation, die das Objekt, den Stack, verändert. Eine glücklichere Wahl ist die Verwendung von `Pop`, denn „push“ und „pop“ sind eingebürgerte Begriffe für die Verwendung eines Stacks. Die Funktion `GetLatchValue()` dürfte einen Wert berechnen, der später in ein Register geschrieben werden muss. Also ist die Bezeichnung irreführend, denn der Benutzer der Funktion erhält nicht den Wert des Registers sondern einen Wert *für* das Register.

Die Wahl des klein geschriebenen „L“ als Variablenname ist gefährlich. In den letzten beiden Zeilen ist es schwierig zu sagen, ob der zuvor gelesene Wert verwendet wird oder eine Eins.

Der Name `LatchValue` ist ein „aktionsloses“ Hauptwort, bezeichnet aber eine Aktion, nämlich das Beschreiben des Registers. Die Wahl des Namens für die letzte Funktion ist ebenfalls irreführend, es wird kein Stack geschrieben. Das Argument der Funktion ist kein Stack sondern ein Element des Stacks. `WriteToStack()` wäre da schon besser. Und wirklich gut ist natürlich `Push()`.

Eine korrigierte Version des obigen schlechten Beispiels könnte wiefolgt aussehen:

```
iOldLatchValue = Pop();                 /* oder PopFromStack() */
iNewLatchValue = GetValueForLatch();
SetLatchValue(iNewLatchValue);
Push(iNewLatchValue);                   /* oder PushOnStack()  */
```

Standardisierung der Namensgebung

Wie können wir nun diese Erfahrungswerte in einen Standard einbringen? Der erste Schritt ist, wie erwähnt, das Festhalten von Fachwörtern, wie in Tabelle 1 gezeigt. Die Liste der Fachwörter sollte auch branchenspezifische Begriffe umfassen und daher wesentlich umfangreicher als das gezeigte Beispiel sein.

Des weiteren müssen wir die Verwendung dieser Begriffe standardisieren.

Regel 8: Verwendung von Fachbegriffen in Bezeichnern

> Die in Tabelle 1 angeführten Fachbegriffe dürfen nur in der beschriebenen Form verwendet werden.

Regel 9: Verwendung von Get/Set/Take in Bezeichnern

> Die Vorsilben `Get`, `Set` und `Take` müssen von einem Hauptwort gefolgt werden, das das von der Operation betroffene Objekt oder seine Komponente bezeichnet.

Dank dieser Regel sind auch die gelegentlich anzutreffenden und sehr irreführenden Funktionsnamen der folgenden Art nicht gestattet.

```
uiNewLatchValue = GetLatchValue();   /* berechne neuen Wert;  */
port_io(LATCH1, uiNewLatchValue);    /* setze den Wert in ... */
```

Die Funktion `GetLatchValue()` holt den neuen Wert nicht vom Register (`Latch`) sondern berechnet ihn, daher ist die Bezeichnung `GetValueForLatch()` viel angebrachter.

Die Wahl von Namen selbst ist schwer zu standardisieren. Wohl aber können wir Richtlinien aufstellen, wie die folgenden Empfehlungen und Vorschriften zeigen.

Empfehlung 10: Worttypen in Bezeichnern für Unterprogramme

> Unterprogramme, Prozeduren oder Funktionen im synchronen Programmfluss sollten Zeitwörter am Beginn des Bezeichners führen, z.B. `StartTimer()`. Eine Ausnahme bilden Bool’sche Funktionen. Asynchrone Routinen sollten im Bezeichner anführen, welches Ereignis sie bearbeiten, z.B. `TimerInterruptServiceRoutine()`.

Begriff	Verwendung	Erläuterung
dummy	Name oder Teil eines Namens für eine Variable.	Dieser Name wird ausschließlich für Variablen verwendet, deren Wert nicht von Interesse ist.
get	Teil eines Namens für ein Unterprogramm.	Abfragen eines Objekts, das Objekt selbst bleibt unverändert. Siehe auch *take* und Regel 9.
ID	Teil eines Namens für eine Variable (Abkürzung für „identifier“); häufig für Nachrichten (messages) oder Prozesse (tasks) verwendet.	Die betroffene Variable ist ein eindeutiger Schlüssel zur Identifikation des Objekts, zu dem diese Variable gehört. Siehe auch *key* und *type*.
key	Teil eines Namens für eine Variable; häufig im Bereich relationale Datenbanken verwendet.	Ein Schlüssel (key) identifiziert einen Eintrag in einer Datenbank (oder einer anderen ein- oder mehrdimensionalen Datenstruktur) eindeutig. Siehe auch *ID*.
pop	Name oder Teil eines Namens für ein Unterprogramm.	Entnehmen des obersten Elements eines Stacks.
push	Name oder Teil eines Namens für ein Unterprogramm.	Einfügen eines Elements in eine eindimensionale Datenstruktur (meist Stack).
set	Teil eines Namens für ein Unterprogramm.	Verändern eines Objekts, siehe Regel 9.
take	Teil eines Namens für ein Unterprogramm.	Abfragen eines Objekts, das Objekt wird dabei in der Komponente des Rückgabewerts der Funktion verändert. Siehe auch *get*.
type	Letzter Teil eines Namens für eine Variable.	Identifiziert den Typ (Klasse) einer Struktur (z.B. Nachricht) eindeutig, es kann aber mehrere Instanzen (z.B. Nachrichten) diesen Typs geben, die ggf. durch eine *ID* unterschieden werden.

Tabelle 1: Erfassung und Definition von Fachbegriffen im Coding-Standard.

Empfehlung 11: Worttypen in Bezeichnern für Bool´sche Unterprogramme

Unterprogramme, Prozeduren oder Funktionen, deren Rückgabewert eine Bool´sche Variable ist, sollten mit Namen bezeichnet werden, die eine Entscheidungsfrage darstellen und daher nur mit ja oder nein bzw. `true` oder `false` zu beantworten sind. Beispiel: `IsInterruptEnabled()`.

Empfehlung 12: Worttypen in Bezeichnern für Variablen

Bezeichner Bool’scher Variablen sollten Eigenschaften beschreiben, z.B. `bModeDirty`. Bezeichner aller anderen Variablentypen sollten Hauptwörter sein, z.B. `iEventNumber`.

Empfehlung 13: Buchstaben als Bezeichner

> Einzelne Buchstaben sollten nicht als Bezeichner für Variablen oder Unterprogramme verwendet werden. Ausnahmen sind Schleifenzähler und Indizes für Felder (arrays). Das kleine „L“ sollte nie als Bezeichner verwendet werden.

Es kann in seltenen Fällen vorteilhaft sein die letzte Empfehlung nicht zu befolgen, wenn ein bekannter Algorithmus implementiert wird und dort einzelne Buchstaben als Variablen oder Funktionsnamen verwendet werden. Etwa wird bei der Definition der Faktoriellen fast immer *n* verwendet und das Blockieren und Freigeben von Semaphoren oft mit `P()` und `V()` bezeichnet.

Wenn ein Name gefunden worden ist, dann sollte es keine Verwechslungsmöglichkeiten mehr geben. Wir fügen daher folgende Empfehlung hinzu

Empfehlung 14: Eindeutigkeit von Namen

> Ein und derselbe Name sollte nicht für verschiedene Objekte verwendet werden. Ein Objekt sollte nur einen Namen haben.

Diese Empfehlung erscheint überflüssig, C++ erlaubt aber beispielsweise einer Variablen zwei oder mehr Namen zu geben. In der natürlichen Sprache werden oft dem selben Objekt verschiedene Namen gegeben: in diesem Artikel etwa wurde bislang von Unterprogrammen, Prozeduren und Funktionen gesprochen, obwohl im bisherigen Kontext kein Unterschied auszumachen war. Solche Begriffsvermischungen haben in Software nichts verloren.

Was bist du?

Die bislang vorgestellten Vorschriften legen keinen lexikalischen Unterschied zwischen Namen für Variablen, Konstanten, Typen und Unterprogrammen fest. Da deren Verwechslung bei hardwarenaher Programmierung (und daher ggf. schwacher automatischer Prüfung) zu schwer erkennbaren Fehlern führen kann, ist die Festlegung von einfachen Unterscheidungsmerkmalen in Form von Regeln vorteilhaft. Eine inkonsequente Anwendung dieser Regeln kann aber auch fatale Folgen haben, wie ein Beispiel noch zeigen wird.

Das einfachste Unterscheidungsmerkmal ist die Groß/Kleinschreibung. Eine unnötige Fehlerquelle bei hardwarenaher Programmierung ist die Verwechslung von Variablen mit Unterprogrammen (deren Einsprungadresse der Compiler oder Assembler als Variable interpretieren könnte). Wir formulieren daher folgende

Regel 15: Schreibweise von Bezeichnern für Unterprogramme

> Unterprogramme, Prozeduren oder Funktionen müssen mit einem Großbuchstaben beginnen und mindestens einen Kleinbuchstaben enthalten. Wenn der zweite Buchstabe groß geschrieben wird, darf der erste kein „T“ sein um Verwechslungen mit einem Typenbezeichner auszuschließen.

Empfehlung 16: Details zur Schreibweise von Bezeichnern für Unterprogramme

> Unterprogramme, Prozeduren oder Funktionen sollten Großbuchstaben nur verwenden um den Beginn von neuen Worten zu signalisieren, wie zum Beispiel in `ComputeInverseMatrix()`.

Schön wäre es, von einem Bezeichner einer Variablen gleich zu erfahren, von welchem Typ die Variable ist. Dies ist zumindest bei Standard-Typen durch die Verwendung einer Vorsilbe

(Präfix) leicht möglich. Erstmals wurde diese Idee in der sogenannten Ungarischen Notation standardisiert [1]. Tabelle 2 gibt einige weit verbreitete Vorsilben an, die mit den Vorsilben der Ungarischen Notation im übrigen nicht mehr sehr viel zu tun haben. Die Tabelle ist in Englisch verfasst, weil dann die Bedeutung der Vorsilben klar wird. Nun müssen wir die Verwendung dieser Tabelle als Regel erfassen:

Regel 17: Schreibweise von Variablen und Konstanten

Bezeichner für Variablen und Konstanten mit mehr als einem Buchstaben müssen mit einem in Tabelle 2 erfasstem Präfix versehen sein. Der dem Präfix folgende Variablenname selbst muss Empfehlung 15 befolgen. Ein dem Präfix folgender Namen für Konstanten muss in Blockbuchstaben geschrieben sein.

Gültige Bezeichner für Variablen und Konstanten sind dann zum Beispiel:

```
fPI        = 3.14159; /* typenbehaftete Konstante */
bDoorOpen  = true;    /* Bool´sche Variable */
uiBaudRate = 1200;    /* ganzzahliger, positiver Wert */
wBaudRate  = 0x04B0;  /* ganzzahliger, pos.Wert, 16 Bit breit */
fSimple    = 1.1234;  /* einfache Gleitkommazahl */
dfDouble   = 4.0e+83; /* Gleitkommazahl höherer Auflösung */
ppiJunk    = 0;       /* ein Zeiger auf einen Zeiger auf einen
                       * ganzzahligen Wert */
```

Prefix	Variable Type
b	Boolean
c	character
d	double precision/width (must be followded by another prefix)
e	enumeration
f	floating point number in the default precision of the processor
h	short integer (uses only half of the default bit length of the processor)
i	integer (default bit length of the processor)
p	pointer (should be followded by another prefix that identifies the type to which the pointer is pointing to)
s	string (the identification of the end of the string is supported by the type itself)
sz	zero-terminated string
t	user defined type
u	unsigned (must be followded by another prefix)
w	word – an unsigned 16-bit entitiy

Tabelle 2: Vorsilben (engl: prefixes) für Variablen.

Um die Unterscheidung selbstdefinierter Typen von Funktionen und Variablen leichter zu machen formulieren wir folgende

Regel 18: Schreibweise von Typennamen

Ein Bezeichner eines benutzerdefinierten Typs muss mit einem großen Anfangsbuchstaben beginnen, mindestens einen Kleinbuchstaben enthalten und mit dem Präfix „T“ versehen werden. Zum Beispiel: `TDatabaseRecord`.

Die vorgestellten Regeln können zur Falle werden, wenn sie nicht konsequent angewandt werden, wie folgendes Beispiel in C zeigt.

```
/* Team A stellt folgenden Parameter zur Verfügung: */
int* iParameter;  /* ist Zeiger, hat aber kein "p" im Präfix.
                   * Ist die Variable, auf die der Zeiger zeigt
                   * Null, so ist ein Modul in den Normal-
                   * Zustand gesetzt. */

/* Team B verwendet diesen Parameter, prüft aber nicht seine
 * Deklaration, weil das Präfix schon eine Integer-Variable
 * definiert. */
iParameter = 0;  /* Annahme: diese Anweisung setzt das Modul in
                  * den Normalzustand. Tatsächlich ist der
                  * Zustand aber undefiniert. */
```

Woher kommst du?

Wer einmal in einem mehrere tausend Zeilen umfassenden Quellcode einen Fehler finden musste und nicht selbst Autor der Software war, weiß, wie schwierig es sein kann, eine bestimmte Definition aufzufinden. Warum dies von Bedeutung sein kann, wurde im obigen Beispiel bereits demonstriert. Die Auffindbarkeit von Definitionen wird wesentlich verbessert, wenn jedes global verwendete Programmelement als Teil des Bezeichners den Ort seiner Definition angibt, wie folgende Regel vorschreibt:

Regel 19: Definitionsort in Bezeichnern globaler Programmelemente

> Ein Bezeichner einer globalen Variable oder eines globalen Unterprogramms muss die Datei identifizieren, in der die Definition der Variablen/des Unterprogramms erfolgt.

Diese Regel ist sehr allgemein gehalten und lässt viele Freiheiten. Der Grund dafür ist die Vielfalt der Programmiersprachen und Programmierstile. Nehmen wir das Beispiel reinen objektorientierten Designs. Wenn die Programmiersprache Objektorientierung unterstützt, dann ist es üblich, jede global sichtbare Klasse in einer Datei gleichen Namens zu definieren. In einem sauberen Design (keine globalen Attribute und Variablen) ist jede weiter Maßnahme zur Identifikation des Definitionsorts überflüssig. Programmieren Sie aber beispielsweise in C oder Assembler, dann kann die Beachtung folgender Empfehlung viel Ärger sparen.

Empfehlung 20: Erfassen des Definitionsorts in nicht-objektorientierten Sprachen

> Jedes gobal sichtbare Programmelement sollte den Namen der Datei (oder eine eindeutige Abkürzung des Dateinamens) in der es definiert wird, im Bezeichner führen und diesen durch ein Unterstreichungszeichen vom Rest des Bezeichners trennen.
> Unterstreichungszeichen sollten ausschließlich diesem Zweck dienen.

Diese Empfehlung ruft zum sparsamen Gebrauch des Unterstreichungszeichens auf, weil damit Namen kürzer werden – viele Compiler unterstützen Namenslängen bis nur maximal 32 Zeichen. Es fehlt allerdings der Hinweis, ob die Kennung des Definitionsortes als Vor- oder Nachsilbe erfolgen sollte. Diese Festlegung kann in einer weiteren Empfehlung gemacht werden, und ggf. für Typen und Unterprogramme verschieden sein.

Rein formal wird streng zwischen einer Deklaration und einer Definition unterschieden. Der Begriff „Definition“ in obigen Vorschriften ist nicht formal zu verstehen: mit dem Ort der Definition ist die Stelle im Programm gemeint, wo der Programmierer den Typ des gesuchten Elements erkennen kann und entsprechende Kommentare vorfindet.

Beispiele für die Verwendung global sichtbarer Programmelemente in C:

```
/* Funktionen, die in der Datei DataSink.c implementiert sind:
 */
bOpen = datasink_is_oximeter_chnannel_open();   /* So nicht! */
bOpen = DataSink_IsOximeterChnlOpen();       /* Viel besser! */
SPEAKER_MAKESOUND(DIALTONE);      /* ein Makro aus Speaker.h */
{
   /* Das Designdokument identifiziert die Datei MsgIntf.h als
    * Definitionsort von Nachrichten, die in zwei Systemen
    * interpretiert werden müssen. Das gegenständliche Programm
    * behandelt die Nachrichten im Modul MsgHnd.c */
   TAlertMessage_MsgIntf tMyAlarmMessage;
   TStatusMessage_MsgIntf tMyStatusMessage;
   MsgHnd_Clear(&tMyStatusMessage);
   ...
}
```

Die verwendete Abkürzung `Chnl` ist gemäß Regel 2 wohldefiniert.

Einsatz in der Praxis

Wenn Sie die vorgestellten Regeln und Empfehlungen in Ihrem eigenen Coding-Standard verwenden, dann sollten Sie einen formaleren Stil wählen, als Sie es hier im Artikel vorfinden. Beispielsweise können Sie im Standard das Kapitel „Namensgebung“ definieren und in Unterkapitel „Generelle Richtlinien“, „Namen für Variablen“, „Funktionsnamen“ u.s.w. teilen. Das erhöht den Wert des Dokuments als Nachschlagewerk, die im Artikel gewählte Reihenfolge ist rein didaktisch begründet und muss nicht unbedingt eine glückliche Wahl für ein verbindliches Dokument sein.

Regeln in Ihrem Coding-Standard sollten mit eindeutigen Bezeichnern versehen sein. Die im Artikel gewählte einfache Nummerierung lässt zwar ein einfaches Umwandeln einer Empfehlung in eine Regel zu, ist aber ungeschickt, wenn neue Regeln in der Mitte des Texts hinzugefügt werden. Ein bessern geeigneter Bezeichner ist etwa **R Nam-Gen 4-2** für die vierte Regel zur Namensgebung, Unterkapitel „Generelles“. Diese Regel wurde zuletzt in Version 2 des Coding-Standards geändert. Empfehlung **E Nam-File 1-1** identifiziert die erste Empfehlung zur Namensgebung von Files und ist seit Version 1 des Coding-Standards unverändert. Diese Art von Bezeichnern lässt sich dann einfach in einem Protokoll für ein Code-Review verwenden, um dem Programmautor aufzuzeigen, welche Regel oder Empfehlung er verletzt hat.

Empfehlenswert ist auch die Standardisierung und Protokollierung von Code-Inspektionen und Code-Reviews mit Hilfe von Checklisten. Der Umfang an Standards und Vorschriften, die den Programmierern zugemutet wird, hängt üblicherweise stark vom Einsatzfeld der Software ab. Die Standards zur Entwicklung von Firmware eines Herzschrittmachers sind sicherlich

umfangreicher als die für eine Windows-Applikation (bei der die Erwartungshaltung bezüglich Softwarequalität ohnehin recht gering ist). Scherz beiseite: auch auf dem PC-Sektor haben Coding-Standards längst Einzug gehalten, die Urform der Ungarischen Notation wurde von einem Mitarbeiter von Microsoft publiziert und im Application Programmers Interface (WIN API) auch weitgehend umgesetzt.

Leider finden sich gute Beispiele für die Missachtung von lexikalischen Standards selbst in weit verbreiteten Programmiersprachen, wenn sie „zu schnell gewachsen" sind, wie C++ eindrucksvoll demonstriert: so kann der Zustand der Klasse `iostream` etwa durch `setstate()` gesetzt werden, was unseren Ansprüchen an einen Standard schon sehr nahe kommt, aber die Breite der Zahlendarstellung wird mit `width()` gesetzt, anstatt mit `setwidth()`. Zu viele Köche verderben den Brei – die Mitglieder des C++ Konsortiums konnten sich offensichlich auf keine einheitliche Namensgebung einigen. Aber auch auf keine einheitliche Schreibweise: die Klasse istream definiert die Funktion `putback()`; „put" und „back" sind in keiner Weise getrennt. In der Standard Template Library wird plötzlich zwischen dem Zeitwort und der Destination ein Unterstreichungszeichen eingeführt und die Funktion `push_back()` definiert. In Sprachen wie Ada oder Modula2, deren Entwicklung viel langsamer vor sich ging und die unter anderem auch deshalb weit weniger verbreitet sind, finden sich derartige Unschönheiten nicht.

Weitere Maßnahmen

Die vorgestellten lexikalischen Regeln sind *ein* wichtiger Baustein eines Coding-Standards. Volle Geltung erlangen sie allerdings erst mit weiteren Standardisierungen, die das Layout, das Design, das Konfigurationsmanagement und auch Elemente von verwendeten Programmiersprachen betreffen [2]. Wir werden in unserem nächsten Artikel die vorgestellten lexikalischen Regeln um Feinheiten in Bezug auf Benennung von Makros und Dateien erweitern und uns ausführlich mit dem Code-Layout auseinandersetzen.

Referenzen

[1] http://msdn.microsoft.com/library/techart/hunganotat.htm
[2] Codierungsstandards in der Software-Entwicklung für Eingebettete Systeme.

Professionelles Codelayout

Tipps zur Verbesserung und Standardisierung des Layouts von Quellcode

Wenn ein professioneller Programmierstil in Form von Regeln und Empfehlungen dokumentiert ist, so sprechen wir von einem Software-Codierungsstandard. Im folgenden Artikel werden Regeln vorgestellt, die das Code-Layout betreffen und wesentlicher Bestandteil eines solchen Standards sind. Die Regeln sind so formuliert, dass sie für eine Vielzahl von Programmiersprachen anwendbar sind. Die Beachtung dieser Vorschriften erleichtert vor allem die Zusammenarbeit im Team und erhöht die Wartbarkeit der Software.

Das Layout von Quellcode ist nicht nur eine Frage persönlichen Geschmacks oder eine Frage des Programmierstils. Ein vereinheitlichtes Code-Layout in einem Softwareprojekt ist eine wichtige Maßnahme zur besseren Interaktion von Teammitgliedern und damit ein unverzichtbarer Bestandteil eines Codierungsstandards [1]. Im folgenden wird eine solche Standardisierung vorgestellt und mit Programmbeispielen illustriert. Weil C die meist verbreitete Sprache im Sektor Eingebettete Systeme ist und das Layout zwangsweise mit der verwendeten Programmiersprache variiert, sind die meisten Programmbeispiele dieses Artikels in C verfasst (und sind damit automatisch auch für C++ anwendbar). Die hier angesprochenen Themen sind aber weitgehend sprachunabhängig und auf alle prozeduralen höheren Programmiersprachen anwendbar, insbesondere auch für Java und Ada. Der Großteil des Artikels ist auch für die Programmierung in Assembler relevant.

Regeln für ein professionelles Layout: das Leerzeichen

In einem früheren Artikel wurden bereits Regeln zur Namensgebung und Zahlendarstellung vorgestellt [2]. Wir werden diesen Satz von Regeln nun erweitern. Einige der Regeln werden erfahrene Programmierer als überflüssig erachten, weil sie diese ohnehin intuitiv anwenden, ohne sich darüber viele Gedanken zu machen. Software wird aber nicht nur von „alten Profis“ geschrieben, es lohnt sich daher, zu definieren, was einen guten Stil im Bereich des Code-Layouts ausmacht. Wir beginnen mit folgenden Festlegungen:

Regel 1: Delimitoren und Ausdrücke

> Delimitoren von Statements (also der Strichpunkt in C) oder Variablen (der Beistrich in C) dürfen nicht unmittelbar einem Leerzeichen folgen, müssen aber von einem Zeilenvorschub, einem Leerzeichen oder weiteren Delimitoren gefolgt werden.

Regel 2: Trennzeichen in zusammengesetzten Datentypen

> Trennzeichen für zusammengesetzte Datentypen (also z.B. der Punkt beim Spezifizieren einer Komponente einer `struct` oder `union` in C) dürfen weder direkt vor einem Leerzeichen stehen noch direkt von einem Leerzeichen gefolgt werden.

Regel 3: Leerzeichen und Zuweisungsoperator

> Der Zuweisungsoperator muss zwischen zwei Leerzeichen oder zwischen einem Leerzeichen und einem Zeilenvorschub stehen.

Regel 4: Leerzeichen und unäre Operatoren

Unäre Operatoren (wie z.B. in C der Adressoperator & oder der Verweisoperator *) dürfen von ihrem Operanden nicht durch ein Leerzeichen getrennt werden.

In C und C++ kann im Fall von Zeigervariablen der Stern-Operator auch zu einem Teil einer Variablendeklaration werden. Ähnliches gilt analog für Typenbezeichner in Ada und Pascal. Um auch hier potenziellen Fehlern vorzubeugen, erweitern wir Regel 4 folgendermaßen:

Regel 5: Unäre Operatoren als Teil von Datendeklarationen

Ein unärer Operator, der in der Bezeichnung einer Variablen vorkommen kann, darf vom Variablenbezeichner nicht getrennt werden.

In vielen gängigen Codierungsstandards gibt es schärfere Vorschriften betreffend der Verwendung des Leerzeichens anstelle der beiden folgenden Regeln. Oft wird beidseitig jedes binären Operators (wie z.B. &&, +, *) ein Leerzeichen gefordert. Die Motivation hinter der hier vorgestellten weniger strikten Handhabung ist die Möglichkeit, Code-Zeilen, die bei Verwendung von aussagekräftigen Bezeichnern tendenziell eher zu lang geraten, auf diesem Weg erforderlichenfalls zu kürzen.

Regel 6: Leerzeichen und binäre Bool'sche Operatoren

Binären Bool'schen Operatoren muss ein Leerzeichen direkt vorangehen und eines direkt folgen.

Regel 7: Leerzeichen und andere binäre Operatoren

Die Leerzeichensetzung um binäre Operatoren darf visuell nicht den Vorrangregeln bei der arithmetischen Auswertung des Sprachkonstrukts widersprechen. Ein Leerzeichen muss entweder auf beiden Seiten des Operators stehen oder auf keiner der beiden Seiten.

Beispiele zu den vorgestellten Regeln sind in Listing 1 angegeben. Wie schon eingangs erwähnt, wenden viele erfahrene Programmierer diese automatisch an, ohne sich je Gedanken gemacht zu haben, wie man diesen Stil formal erfassen könnte.
Die Definition von Regeln und deren konsequente Anwendung innerhalb eines Projektteams sorgt nicht nur für ein einheitliches und sauberes Erscheinungsbild des Gesamtprojektes. Das Auffinden von Referenzen mit Hilfe von Editorsuchfunktionen wird wesentlich erleichtert, speziell bei der Wartung von Code, dessen Autor man nicht selbst ist.

```
/* Listing 1: Missachtung und Anwendung der Regeln 1 bis 7
 *            Das gezeigte C-Programm is syntaktisch richtig! */

#ifdef SCHLECHTES_LAYOUT

struct TPaar {int iX;int iY;} tPaar;   /* Regel 1 verletzt */
int iWert1 ;                           /* Regel 1 verletzt */
int iWert2 = 1,iWert3 = 2;             /* Regel 1 verletzt */

tPaar. iX = 12;                        /* Regel 2 verletzt */
tPaar.iY =tPaar.iX + 19;               /* Regel 3 verletzt */
iWert1 = iWert2 ++ + ++ iWert3;        /* Regel 4 verletzt */
{
   int* piWert2 = &iWert1,             /* Regel 5 verletzt */
        iScheinbarZeigerVariable;
}
assert(iWert1!=iWert2);                /* Regel 6 verletzt */
iWert3 = iWert1 * iWert2+3;            /* Regel 7 verletzt */
```

```
iWert3 = iWert1 *iWert2 + 3;           /* Regel 7 verletzt */

#else

struct Tpaar
{
   int iX;
   int iY;
} tPaar;                               /* OK! */
int iWert1;                            /* OK! */
int iWert2 = 1, iWert3 = 2;            /* OK! */
tPaar.iX = 12;                         /* OK! */
tPaar.iY = tPaar.iX + 19;              /* OK! */
iWert1 = iWert2++ + ++iWert3;          /* OK! */
{
   int *piWert2 = &iWert1,             /* OK! */
       iKeineZeigerVariable;
}
assert(iWert1 != iWert2);              /* OK! */
iWert3 = iWert1*iWert2 + 3;            /* OK! */
iWert3 = iWert1 * iWert2 + 3;          /* OK! */

#endif
```

Listing 1: Demonstration der Regeln 1 bis 7.

Klammernsetzung

Klammern sind wohl eine der meist verwendeten Elemente vieler Programmiersprachen. In C und verwandten Sprachen definieren runde Klammern die Reihenfolge von Auswertungen in arithmetischen Ausdrücken, eckige Klammern enthalten Indizes für Feldvariable, geschwungene Klammern definieren Programmblöcke. In C werden spitze Klammern nur vom Preprozessor verwendet, in C++ werden diese auch für Templates verwendet.

Regel 8: Block Definitionen

> Klammern (oder Schlüsselworte) zur Blockdefinition müssen in einer separaten Zeile stehen und nach dem vorhergegangenen Ausdruck linksbündig ausgerichtet sein.

Diese Regel bedeutet für den C-Programmierer die Festlegung auf die erste der in Listing 2 gezeigten drei gängigen Einrückungs-Strategien. Die Wahl fällt auf diese Strategie, da bei ihr am einfachsten Anfang und Ende eines Programmblocks zu erkennen sind.

Für runde und eckige Klammern definieren wir drei etwas komplexere Vorschriften. Das Klammern-Layout entstammt den Programmbeispielen in den UNIX man-pages. Die Anwendung dieser Vorschriften wird in Listing 3 illustriert.

Empfehlung 8: Klammernsetzung in Ausdrücken und Formeln

> Eine öffnende eckige oder runde Klammer muss einem Leerzeichen, Zeilenvorschub, einer anderen Klammer oder einem Funktions- bzw. Variablennamen folgen und darf nicht direkt von einem Leerzeichen gefolgt werden. Eine schließende eckige oder runde Klammer darf nicht direkt hinter einem Leerzeichen stehen und muss direkt von einem Trennzeichen, Delimitor oder einer weiteren Klammer gefolgt werden.

```
if (i == 1)
{
   /* das ist die in Regel 8 definierte Strategie */
   printf("Codierung Standards sind wichtig.");
}

if (i == 2)
   {
   /* auch diese Form sieht man gelegentlich */
   printf("Hier ist es schwerer vom Blockbegin sofort"
          "auf das Blockende zu schließen.");
   }

if (i == 3) {
   /* diese Form nimmt ein Schlüsselwort als Blockbeginn */
   printf("Die vertikale Ausrichtung des Blockende-Zeichens"
          "passt nicht zum Blockbeginn-Zeichen.");
}
```

Listing 2: Veranschaulichung von Regel 8.

```
/* Klammersetzung gemäß der im Artikel definierten
 * Richtlinien */

void Stop();                        /* Klammer folgt Klammer */
void Stop(void);        /* keine Leerz. bei Unterprogrammen */
i = GetXY(&iX, &iY);    /* Klammern- u. Leerzeichensetzung. */
for (;; i++);               /* "for" kontrolliert den Fluss */
switch (iInputValue)         /* Änderung d. Programmflusses */
while (k > 30)            /* "while" kontrolliert den Fluss */
return (iVar – 1);          /* Änderung des Programmflusses */
exit(-1);             /* "exit" ist kein Schlüsselwort in C */
#define SIN(x) sin(x)  /* in C keine andere Wahl bei Makros */
if (i == 1) exit(0);      /* Programmfluss, Funktionsaufruf */

/* nun zwei Beispielformeln */
i = ((j * (k – 3)) & MASK) >> piShiftValue[i].right;
bRetVal = (i-12)/sin(i) – pfMagicTable[i+1][pfTable2[j]];


/* andere gängige Klammersetzungsvorschriften und deren
 * Relation zu den hier festgelegten Richtlinien */

/* Stil der Beispiele in MSVC 6.0; Verletzung von Empf. 8 */
if ( i == 1 ) exit( 0 );

/* folgender Stil hat Nachteile bei unkonventionellen
 * Implementierungen der C-Standardbibliothek. Wenn
 * exit() ein Makro ist, dann kann die folgende Zeile
 * nicht übersetzt werden. Verletzung von Regel 9. */
if (i == 1) exit (0);

/* häufig zu sehender Stil; Verletzung von Regel 10:
 * Die von "if" erwirkte Programmverzweigung unterscheidet
 * wenig von einem Funktionsnamen. */
if( i == 1 ) exit( 0 );
```

Listing 3: Beispiele zur Klammersetzung.

Regel 9: Klammern für den Aufruf von Unterprogrammen

Vor öffnenden Klammern, die Parameter vom Bezeichner eines Unterprogramms trennen, darf kein Leerzeichen stehen.

Regel 10: Klammern und Schlüsselworte

Zwischen einem Schlüsselwort, das den Programmfluss ändern kann und einer öffnenden Klammer muss ein Leerzeichen stehen.

Die bisher vorgestellten Regeln helfen einerseits, Fehlinterpretationen von komplexen Ausdrücken zu vermeiden und andererseits durch die Festlegung der Verwendung des Leerzeichens, die Textsuche effizienter einzusetzen.

Horizontale Ausrichtung

Die horizontale Ausrichtung (Einrückung) von Programmblöcken ist unverzichtbarer Bestandteil guten Layouts in strukturierten Programmiersprachen und kann auch die Interpretation von Assemblerprogrammen deutlich vereinfachen. Der Zweck der horizontalen Ausrichtung ist es, die Begrenzungen von Programmblöcken zu verdeutlichen. So lässt sich etwa leichter erkennen, welche Instruktionen von einer Verzweigung betroffen sind, übersprungen werden oder Teil einer Schleife sind. Die Beachtung der im folgenden aufgestellten Regeln vereinfacht das Erkennen dieser Blockstrukturen und erhöht damit die Übersichtlichkeit des Quellcodes.

Regel 11: Einrückung und Einrückungsszeichen

Programmblöcke müssen durch Einrückung erkennbar sein. Diese horizontale Ausrichtung muss mit dem Leerzeichen erfolgen.

Die horizontale Ausrichtung von Programmcode mit dem Tabulatorzeichen kann zu unglaublichen Schwierigkeiten führen, wenn fallweise ein anderer Editor verwendet wird oder unter unterschiedlichen Betriebssystemen editiert wird – es ist gar nicht so selten, das der Debugger auf einer anderen Plattform läuft. Das gilt auch für Produkte vom gleichen Hersteller: ein mit dem Microsoft Visual C++ Editor erstelltes Programm kann in der Ansicht des Windows Notepads völlig unleserlich sein. Gute Editoren erlauben trotzdem die Verwendung des Tabulators und übersetzen das Drücken der Tabulatortaste in eine einstellbare Anzahl von Leerzeichen.

Regel 12: Einrückungstiefe

Die Einrückungstiefe muss 3 Zeichen sein.

Warum gerade drei? Viele Programmierer halten zwei Zeichen für zu wenig und den von vielen Betriebssystemen verwendeten Standard von acht Zeichen bei weitem für zu viel. Drei oder vier Zeichen entsprechen der Standardeinstellung vieler Editorprogramme, können noch eindeutig als Einrückung erkannt werden und verschwenden nicht unnötig Platz.

Regel 13: Globale und lokale Einrückung

Code in globalem Kontext (z.B. die Definition global sichtbarer Funktionen) darf nicht eingerückt sein. Gültige Ausdrücke inmitten eines Blocks müssen linksbündig zum vorhergehenden Ausdruck ausgerichtet werden.

Empfehlung 14: Einrückung von Sprungmarken

Eine Sprungmarke (Label) sollte eine einfache negative Einrückung zum vorhergehenden Code aufweisen.

Empfehlung 15: Kommentar am Blockende

Ist ein Programmblock länger als 20 Zeilen, so sollte am Blockende ein Kommentar den Grund der Blockdefinition erläutern.

Bei der Programmierung in Assembler lohnt es sich, zum besseren Auffinden von Sprungmarken Regel 13 zu ändern und eine Mindesteinrückung für Instruktionen zu fordern, wie am Ende von Listing 4 zu sehen ist. Es ist in Assembler auch schwierig, für jeden Sprung eine Einrückung vorzusehen, weil z.B. Fehlerbehandlungen zu unzähligen Sprüngen führen können. In einem derartigen Fall ist die Verzweigungsstruktur meist wesentlich komplizierter als in einer Hochsprache und die Einrückungen wären dann eher kontraproduktiv. Empfehlung 14 hilft beim Verfolgen von Sprungmarken und verzerrt dabei nicht den Kontext eines Sprungs. Ein Beispiel für eine „negative Einrückung" ist ebenfalls am Ende von Listing 4 zu finden, genauso wie die Kommentierung des Blockendes.

Vertikale Ausrichtung und Kommentarzeilen

Der Beginn einer neuen Zeile oder die Verwendung von Leerzeilen kann die Leserlichkeit von Programmcode deutlich erhöhen. Hand in Hand mit dem überlegten Umgang mit dem Zeilenvorschub geht die großzügige Vervollständigung des Codes mit Kommentaren. Dabei ist es vorteilhaft, Kommentare, die ein Statement in der selben Zeile erläutern, von jenen zu unterscheiden, die eine eigene Zeile beanspruchen.

Empfehlung 16: Ganzzeilige Kommentare und Leerzeilen

Kommentare, die zumindest eine eigene Zeile beanspruchen, sollten beschreiben, was in den folgenden Programmzeilen bis zur nächsten Leerzeile passiert und linksbündig nach diesen ausgerichtet sein. Ausnahme: Funktionsbeschreibungen, siehe Regel 19.

Empfehlung 17: Blickfang-Kommentare und Leerzeilen

Kommentare, die die Bedeutung mehrerer Programmzeilen beschreiben, welche auch durch einzelne Leerzeilen getrennt sein können, sollten durch Verwendung von speziellen Kommentarzeichen die Aufmerksamkeit des Lesers erwecken. Diesen Blickfang-Kommentaren sollten innerhalb einer Funktion zwei aufeinanderfolgenden Leerzeilen vorangehen, ansonsten drei Leerzeilen.

Regel 18: Leerzeilen zwischen Funktionen bzw. Unterprogrammen

Funktionen (oder in Assembler: global sichtbare Einsprungadressen) bzw. deren Beschreibung müssen vom vorhergehenden Quellcode durch drei Leerzeilen getrennt sein.

Beispiele zur Beachtung dieser Richtlinien sind in Listing 4 zu sehen. Es macht Sinn, den Blickfang-Kommentar zu standardisieren. In Regel 18 wird die „Beschreibung" der Funktion oder des Unterprogramms erwähnt. In der Tat sollte zu jeder Funktion eine Beschreibung ihrer Schnittstelle existieren. Das Format einer derartigen Funktionsbeschreibung könnte (und sollte) ebenfalls Bestandteil eines Codierungsstandards sein. Etwa die verpflichtende und einheitliche Deklaration von Eingabe-, Ausgabe- und Ein/Ausgabeparametern; das Erwähnen aller verwendeten globalen Variablen, die Beschreibung des Rückgabewerts und allenfalls Fehlerfälle. Am Besten ist es, ein Beispiel der standardisierten Funktionsbeschreibung in elektronischer Form zur Verfügung zu stellen und im Codierungsstandard darauf zu verweisen. Listing 4 zeigt eine informelle Beschreibung der Funktionen mit einem einheitlichen Kommentar als visuelle Hilfe, um den Beginn einer neuen Funktion leichter zu erkennen. Wir formulieren unsere nächste Regel gemäß diesem Beispiel.

```
#ifdef _GNUC_

#include "listing4.h"



/*
 *  Blickfang 1: 3 Leerzeilen davor, da ausserhalb
 *  von Funktionen (Empfehlung 17)
 */

int ErsteVariablendeklaration;
int ZweiteVariablendeklaration;



/************************************************************/
int main()
/************************************************************/
{
   DieFunktion(45);
   return 0;
}



/************************************************************/
static void DieFunktion(int iParameter)
/*
 * Diese Funktion demonstriert unter anderem die neg.
 * Einrückung von Sprungmarken. Der Parameter iParameter
 * ist ein reines Anschauungsobjekt ohne Bedeutung.
 ************************************************************/
{
   iParameter++;


   /*
    *  Blickfang 2: 2 Leerzeilen davor, da innerhalb
    *  einer Funktion (Empfehlung 17)
    */

   if (iParameter > 10)
   {
      iParameter++;
      if (iParameter > 20) goto Label;
      (void) WeitererProgrammcode();
   Label:
      (void) WeitererProgrammcode();
      if (iParameter > 100)
      {
         /* Nun passiert blablabla ... */
         WeitererCode1();  /* erster Schritt zu blabla  */
         WeitererCode2();  /* zweiter Schritt zu blabla */
         WeitererCode3();  /* dritter Schritt zu blabla */
      }
   } /* iParameter > 10 */
   return;
} /* DieFunktion() */
```

```
/************************************************************/
static int WeitererProgrammcode(void);
/*
 * Diese Funktion macht nichts. Im Fehlerfall ist der
 * Rückgabewert (-1), ansonsten 0.
 ************************************************************/
{
   /* tue nichts */
   int i;
   for (i = 0; i < 0; i++)
   {
      return –1;
   }
   return 0;
}


#else /* _GNUC_ */



////////////////////////////////////////////////////////////
DieFunktion:
//
// Assemblerbeispiel. Die Sprungmarke DieFunktion ist nicht
// eingerückt. Also ist diese Funktion, im Gegensatz zur
// obigen C-Version, global sichtbar.
// Parameter x wird in Register r0 übergeben.
// Weitere verwendete Register: r1
////////////////////////////////////////////////////////////
   r1 = 10;
   comp(r0, r1);                  // Parameter x steht in r0
   if le jump DieFunktionLE10;
      r1 = 20;           // Einrückung eines Blocks wie in C
      comp(r0, r1);
      if le jump Label;  // diesmal keine Einrückg., wie in C
      nop;                        // weiterer Programmcode
   Label:                         // negative Einrückung
      nop;                        // weiterer Programmcode
      r1 = 100;
      comp(r0, r1);
      if le jump DieFunktionLE100;
         nop;                     // weiterer Code 1
         nop;                     // weiterer Code 2
         nop;                     // weiterer Code 3
   DieFunktionLE100:
   DieFunktionLE10:
   rts;  // DieFunktion

#endif /* _GNUC_ */
```

Listing 4: Beispiel zu Sprungmarken und Kommentierung von großen Blöcken sowie zur Verwendung von Leerzeilen.

Regel 19: Beschreibungen von Funktionen und Unterprogrammen

Funktionsbeschreibungen müssen alle Parameter, Fehlerfälle und den Rückgabewert der Funktion beschreiben. Bei ambivalenter Interpretationsmöglichkeit muss erwähnt werden,

ob ein Parameter als Eingabewert, Ausgabewert oder für beide Zwecke verwendet wird. Funktionsbeschreibungen müssen dem in Listing 4 gezeigten Layout folgen.

Wenn Sie eine Bibliothek erstellen, dann wird im Regelfall dem Benutzer der Bibliothek nur mehr eine Header-Datei als einziger Teil des Quellcodes zugänglich gemacht. Daher sollten Sie Funktionen auch dort beschreiben und vor allem auf die für deren Verwendung relevanten Randbedingungen eingehen. Interne Funktionen (etwa Funktionen mit dem Attribut `static` in C oder `private` member functions in C++) müssen dem Benutzer der Bibliothek nicht bekannt sein, deren Beschreibung sollte daher nur in der Datei mit der Implementierung der Funktion stehen. Daher lautet die nächste Empfehlung:

Empfehlung 20: Ort der Funktionsbeschreibung

Beschreibungen von internen Funktionen sollten am Ort ihrer Definition (an der Stelle der Implementierung) zu finden sein.
Beschreibungen extern sichtbarer Funktionen (die Bedeutung ihrer Parameter, Rückgabewerte und Randbedingungen für die Funktionsverwendung) sollten am Ort der Funktionsdeklaration (in einer Header-Datei) zu finden sein.

Um das Layout von Quellcode qualitativ aufzuwerten bedarf es nicht nur einer guten Struktur zum Auffinden von Funktionen und Blöcken. Überlange Zeilen sind nicht nur bei machen Editoren bösartige Fallen[1], sie zerstören auch das Layout beim Drucken von Quellcode nachhaltig. Daher ist es unabdingbar, sich an eine maximale Zeilenbreite zu halten. Die Länge dieser Zeilenbreite sollten Sie an den in Ihrer Firma eingesetzen Editor anpassen, insbesondere an dessen Angebot an skalierbaren, nicht proportionalen Schriftarten sowohl für den Bildschirm als auch für die Druckausgabe. Editor und Programmausdruck müssen einen nicht proportionalen Zeichensatz verwenden, da alle Vorschriften zur horizontalen Ausrichtung auf dieser Eigenschaft aufbauen und nur so das Zählen von Leerzeichen in Strings möglich ist.

Regel 21: Maximale Zeilenbreite von Quellcode

Jede Zeile des Quellcodes darf maximal 90 Zeichen umfassen. Ausdrücke, die diesen Schwellwert überschreiten, müssen auf mehrere Zeilen aufgeteilt werden.

Regel 21 ist eine der wichtigsten der in diesem Artikel vorgestellten Richtlinien und gleichzeitig die kontroversiellste. Regeln zur Namensgebung sollen uns einerseits dazu bringen, aussagekräftige und damit zumeist auch längere Namen zu wählen. Kommentare und die zuvor vorgestellten Regeln zur Verwendung des Leerzeichens machen Programmzeilen ebenfalls länger als unbedingt notwendig. Andererseits verlangt Regel 21 eine Beschränkung der Zeilenlänge und damit entweder einen Kompromiss oder die Aufteilung einzelner Ausdrücke auf mehrere Zeilen. Damit diese Aufteilung nicht zu einer unbeholfenen Notlösung wird, formulieren wir analog zu Regel 7 die folgende

Empfehlung 22: Zeilentrennung und Operatoren

Die Trennung eines Ausdrucks sollte an der Stelle eines Operators erfolgen, der eine keine höhere Priorität in der arithmetischen Auswertung hat als der vorhergehende oder folgende Operator.

[1] Beispiel: Weder der Editor noch der Debugger von MSVC 6.0 macht den Programmierer darauf aufmerksam, dass irrtümlich eine zweite Instruktion weit ans Ende einer Zeile plaziert wurde.

Die folgende Empfehlung verhindert im Regelfall keine Fehlinterpretationen, sondern sorgt nur für ein „sauberes Erscheinungsbild“ des Quellcodes.

Empfehlung 23: Zeilentrennung und Listen

> Die Zeilentrennung bei Auflistungen von Daten (etwa die Initialisierung einer Feldvariable oder die Liste der Parameter bei einem Funktionsaufruf) sollte nach dem Trennungszeichen erfolgen. Der neue Eintrag in der neuen Zeile sollte linksbündig nach dem ersten Eintrag der Liste ausgerichtet sein.

Bei Dezimalzahlen kann es auch günstig sein, sich bei der vertikalen Ausrichtung am Dezimalpunkt zu orientieren anstatt einfach linksbündig zu formatieren. Dementsprechend könnte man Empfehlung 23 auch umformulieren. Zu guter Letzt wollen wir zum Thema Zeilenlänge noch erwähnen, dass das in Regel 19 geforderte Format der Funktionsbeschreibung und ggf. auch ein Datei-Header am besten exakt die maximal zulässige Zeilenlänge verwendet um deren Einhaltung beim Programmieren zu erleichtern, sofern ein Editor dies nicht unterstützt.

Der Programmkopf

In jeder Quelldatei sollte mit Hilfe eines standardisierten Datei-Headers dokumentiert sein, wer sie zu welchem Zweck schrieb. Ein firmenspezifischer Codierungsstandard verweist hier am Besten auf Vorlagen in elektronischer Form, die allen Programmierern zugänglich sind. Welche Elemente ein solcher Datei-Header beinhalten soll, ist Geschmackssache, die unten angeführte Liste bietet eine breite Auswahl. Am besten ist eine automatische Aktualisierung von möglichst vielen Elementen bei Verwendung einer computerunterstützten Versionskontrolle.

- Copyright Statement mit Erwähnung der Firma, des Erstellungsdatums und des Datums der letzten Änderung
- Name des Projekts, für das die Quelldatei erstellt wurde
- Verwendungszweck der Datei (des Moduls)
- Querverweis auf ein Design-Dokument oder Referenzen zu anderer verwendeter Literatur
- verantwortlicher Programmierer
- verantwortlicher Tester, Teststatus
- verwendeter Editor
- verwendetes CASE-Tool oder Dokumentations-Tool
- verwendeter Compiler
- verwendetes Lint-Programm
- aktuelle Versionsnummer und, ganz wichtig, die Revision History.

Es kann nicht oft genug betont werden, wie wertvoll eine richtig geführte Revision History für das Arbeiten mit der Quelldatei sein kann. Sie ist ein Überblick über den Werdegang der Datei, wer wann was und *warum* geändert hat. Die entscheidende Information ist das *warum*, denn wann was geändert wurde kann jederzeit mit einem elektronischen Versionsvergleich auch festgestellt werden. Den Grund für die durchgeführten Änderungen zu kennen kann viel Zeit und Ärger sparen, insbesondere dann, wenn man später in Versuchung kommt, eine Änderung wieder zu revidieren, deren genaue Ursache nicht mehr bekannt ist.

Sprache, Werkzeuge & Design

Die vorgestellten Regeln zum Layout von Quellcode, zu Zahlenformaten und zur Namensgebung sind weitgehend sprachunabhängig [2]. In Listing 5 können wir aber sehen, dass die Sprache C (und viele andere Sprachen auch) noch einige Schlupflöcher durch unser Regelwerk findet. Beide in Listing 5 gezeigten Fallen haben mit dem Layout zu tun und verletzen keine der präsentierten, sprachunabhängigen Regeln. Um solch schlecht strukturierten C-Code zu unterbinden, müssen wir ganz spezifische, sprachabhängige Regeln aufstellen. In unserem nächsten und gleichzeitig letzten Artikel zum Thema Codierungsstandards werden wir genau das tun.

```
i = 0;
while (i < 1200);     /* FALLE: Warteschleife is endlos */
   i = i + 1;

if (k <= i)
   if ( k == i)
      printf("Gleichheit");
else
{
   /* FALLE: hier gilt k < i und nicht k > i */
}
```

Listing 5: Sprachabhängige Fallstricke bei der Ausrichtung von Code.

Neben der computerunterstützten Versionskontrolle haben weitere Werkzeuge eine Schnittstelle zum Codierungsstandard. Je spezifischer eine sprachabhängige Regel des Standards ist, desto leichter kann man sie automatisch überprüfen. In der Tat erkennt ein gutes, kommerzielles Lint-Tool beide Fallen in Listing 5 und das gratis erhältliche LCLint [3] erkennt immerhin die Endlosschleife. LCLint erlaubt übrigens auch in beschränktem Rahmen eine designabhängige Überprüfung des Quellcodes. Das Programmdesign ist untrennbar mit der verwendeten Programmiersprache verknüpft. Unser Folgeartikel wird Designrichtlinien für die C-Programmierung vorstellen, die einfach in einen Codierungsstandard eingegliedert werden können.

Die breiteste Schnittstelle zwischen einem Entwicklungswerkzeug und den Layout-Vorschriften eines Codierungsstandards hat im allgemeinen ein Designprogramm, das automatisch Quellcode erzeugen kann. Wenn Sie ein neues CASE-Tool benutzen, müssen Sie unter Umständen das im Codierungsstandard festgehaltene Layout gründlich überholen oder das Tool entsprechend anpassen.

Referenzen

[1] Codierungsstandards in der Software-Entwicklung für Eingebettete Systeme.
[2] Ein guter Programmstil will standardisiert sein.
[3] LCLint User´s Guide, `http://lclint.cs.virginia.edu/`.

Fehlervermeidung bei der Erstellung von C-Programmen

Die Programmiersprache C ist für viele Anwendungen im Bereich Eingebettete Systeme nicht wegzudenken. Für eine Vielzahl von Prozessoren erlaubt sie nahezu alles zu implementieren, was auch in Maschinensprache möglich ist. Der Preis für diese Flexibilität ist das geringe Sicherheitsniveau beim Programmieren. Der folgende Artikel beschreibt Regeln, deren Befolgung dieses Niveau hebt ohne die Mächtigkeit von C zu bescheiden. Weiters erleichtern sie die Interaktion von Programmierern und sollten somit Teil eines Codierungsstandards sein.

In den frühen Siebzigern arbeitete Dennis Ritchie in den heute zu Lucent gehörenden Bell Laboratories an der Entwicklung des Betriebssystems UNIX. Zu diesem Zweck benötigte er eine Sprache, die kompakten, schnellen Code erzeugen ließ und die Hardware effizient bedient. Zu dieser Zeit wurden diese Kriterien nur von Assembler zuverlässig erfüllt. Doch Assembler ist prozessorabhängig und UNIX sollte auf einer Vielzahl von Plattformen laufen. Daher wurde C geschaffen, eine strukturierte, prozedurale (prozessorunabhängige) Hochsprache, die die Implementierung aller erdenklichen Tricks des einfachen Von-Neumann-Rechners zulässt. Wenn man die Tricks allerdings nicht ganz beherrscht, dann ist es in kaum einer anderen Sprache so leicht sich selbst Fallen zu stellen wie in C, der heute wichtigsten höheren Programmiersprache im Bereich Eingebettete Systeme. Die strukturierte Sprache C erlaubt nicht nur die Erzeugung von nahezu so effizientem Code wie Assembler, sondern gestattet auch einen sagenhaft unleserlichen und unstrukturierten Programmstil. Ein schlechter Programmstil verleitet zu Fehlinterpretationen bei der Wartung von Software, erschwert Modultests und Portierung, kostet ein Unternehmen daher Zeit und Geld. Abhilfe schaffen sogenannte Codierungsstandards [1,2,3]. Die folgenden Vorschriften könnten Teil eines derartigen C-Standards sein.

Vom Umgang mit dem Pre-Processor und Dateien

Ein mächtiges Werkzeug für die Übersetzung von C-Code ist der Pre-Processor, dessen Arbeit schon beendet ist, bevor der Compiler selbst startet. Der sachgemäße Umgang mit seinen Befehlen erleichtert die Portierung von Code.

Regel 1: Pfadangabe in Include-Dateien

Die Angabe eines Pfades im `#include` statement muss relativ zum Projektverzeichis sein. Absolute Pfadangaben sind nicht gestattet.

Regel 2: Allgemeine und projektspezifische Include-Dateien

Projektspezifische Include-Dateien müssen im `#include` statement zwischen Doppelhochkomma stehen. Spitze Klammern sind für C-Runtime-Libraries und andere projektübergreifende Dateien reserviert.

Regel 3: Header und Implementierung

Der Name einer Header-Datei darf sich vom Namen der Implementierungsdatei nur in der Datei-Endung unterscheiden.

Die oben vorgestellten Vorschriften erlauben das einfache Wechseln des Projektverzeichnisses und erleichtern das Auffinden von Dateien und dazugehörigem Code. Um C-Code auch in C++ verwenden zu können und wiederholtes Parsen von Code zu verhindern fomulieren wir

Empfehlung 4: Kapselung von Header-Dateien

Die Deklarationen jeder Header-Datei sollten zwischen den in Listing 1 gezeigten Pre-Processor-Anweisung stehen.

```
#ifndef DATEINAME_H
#define DATEINAME_H

#ifdef __cplusplus
   extern "C" {
#endif

/* hier beginnen die Deklarationen der Header-Datei. */

/* Demo zu Regel 7: */
#if (CPU == ADSP2181)
typedef int TInt16;
#elif (CPU == INTEL486)
typedef short int TInt16;
#else
#error Macro CPU has an unexpected value!
#endif

#ifdef DEBUG
#undef ASSERT
#define ASSERT(a) assert(a)
/* Ein Tippfehlern im Makronamen wird nicht erkannt. */
#endif

/* hier enden die Deklarationen der Header-Datei. */

#ifdef __cplusplus
   }
#endif

#endif /* DATEINAME_H */
```

Listing 1: Beispiel zur Empfehlung 4 und Regel 7.

Der Funktionsumfang des Pre-Processors reicht aber deutlich über das Einfügen von Include-Dateien hinaus. Er gestattet nahezu unbegrenzte textuelle Ersetzung mit Hilfe von Makros. Makros können Parameter benötigen und damit Funktionen imitieren. Sie unterscheiden sich aber in jedem Fall von Funktionen, wie später gezeigt wird. Um C-Makros deutlich von anderen Sprachelementen zu unterscheiden verlangen wir eine reservierte Schreibweise.

Regel 5: Schreibweise von Makros

Namen für Makros dürfen nur Großbuchstaben und das Unterstreichungszeichen enthalten.

Der Pre-Processor gestattet auch die bedingte Übersetzung von Code. Von dieser Fähigkeit sollte aber nur sehr sparsam Gebrauch gemacht werden. Manche Autoren raten sogar gänzlich von deren Verwendung ab [4]. Tippfehler bei der Schreibweise eines Makros können zu unerwarteten Ergebnissen führen, der Quellcode wird schwer leserlich. Daher fordern die folgenden Regeln rigorose Tests und Dokumentation.

Regel 6: Dokumentation bedingter Übersetzung

Alle Makros zur bedingten Übersetzung müssen im Datei-Header und im Designdokument beschrieben werden.

Regel 7: Wert eines Makros

Erwartet eine Quelldatei, dass ein Makros einen bestimmten Wertebereich hat, so muss ein Compilerfehler erzeugt werden, wenn es ein einen ungültigen Wert aufweist.

Ein Beispiel zu Regel 7 ist in Listing 1 zu finden: der Compiler stoppt, wenn das Makro `CPU` einen unerwarteten Wert hat. Hat ein Makro keinen überprüfbaren Wert, so ist diese Technik nicht möglich. Auch dazu sehen wir im Folgenden ein Beispiel. In Listing 1 wird einfach geprüft ob `DEBUG` definiert ist, oder nicht. Ein Tippfehler im Bezeichner des Markso wird nicht autmatisch erkannt. Listing 1 könnte in UNIX durch folgende Anweisung übersetzt werden:

```
cc -c dateiname.c -DDEBUG
```

Genauso gut übersetzt aber die folgende Anweisung mit Tippfehler das Programm, allerdings ohne Debug-Eigenschaften.

```
cc -c dateiname.c –D_DEBUG_
```

Verlangen wir nun in einer neuen Regel *immer* einen Makrowert, so werden derartige Tippfehler abgefangen. Beide zuvor gezeigten Übersetzungsversuche würden fehlschlagen. Je nach Implementierung könnte zum Beispiel die Anweisung

```
cc -c dateiname.c –DDEBUG=1
```

zum gewünschten Ergebnis führen und es müsste in Listing 1 der Wert des Makros `DEBUG` überprüft werden anstelle der Tatsache der Definition. Bei der Einführung einer solchen verschärften Makro-Regel könnte man Regel 6 in eine Empfehlung umwandeln.

Es ist weit verbreitete Praxis die Einrückung von Zeilen von Pre-Processor-Anweisungen unbeeinflusst zu lassen, wie in Listing 1 bei den Makros `DEBUG` und `CPU` gezeigt. Gleichzeitig findet man aber eine Einrückung der `extern`-Deklaration in nahezu jeder Header-Datei, die das Makro `__cplusplus` prüft. Dieses Thema könnte Gegenstand einer weiteren Regel sein.

Fehlervermeidung bei der Steuerung des Programmflusses

In einem vorangegangenen Artikel zum Thema Codelayout wurde gezeigt wie leicht man beim Lesen eines Programms in die Irre geführt werden kann, wenn die horizontale Ausrichtung der Programmzeilen falsche Blockstrukturen vortäuscht [3]. Um derartigen Fehlern nicht nur mit Layout-Maßnahmen zu begegnen, beachtet man am besten

Regel 8: Geschwungene Klammern beim if-statement

Ist ein `if`-statement länger als eine Zeile lang, so müssen die Blöcke für jeden möglichen Programmpfad mit geschwungenen Klammern gefasst werden.

Um den gegenseitigen Ausschluss bei zusammenhängenden `if`/`else`-Folgen zu betonen, notieren wir

Regel 9: Blockstruktur bei „else if“

Wenn mehrere if-statements durch else gekoppelt sind, dann müssen diese dem Layout von Listing 2 gehorchen.

```
if (4 == x) printf("Einzeiliges if, keine geschw. Klammern");
/* Netter Trick am Rande:
 * Beachten Sie, wie obenstehendes IF-statement durch die
 * Wahl der Reihenfolge von Konstanter und Varibler den
 * verbreiten Fehler "if (x = 4) ... " verhindert. */

if (iRegel == 8)
{
   /* "if"-Anweisung ist länger als 2 Zeilen, daher muss
    * sowohl dieser Block, also auch ... */
}
else
{
   /* ... der "else"-Block in geschwungene Klammern gefasst
    * sein. */
}

if (iRegel == 9)
{
   /* Fall 1. */
}
else if (x > 4711)
{
   /* Fall 2. */
}
else if (x == 12)
{
   /* Fall 3. */
}
else
{
   /* Fall 4.
    * Die einzelnen Fälle sind wie in einem switch-statement
    * auf gleichem Einrückungs-Niveau. Der Zeilenbeginn mit
    * "else if" erinnert, dass die Blöcke sich aber gegen-
    * seitig ausschließen. Wenn "else" und "if" durch einen
    * Zeilenvorschub getrennt sind, ist dies nicht mehr sofort
    * ersichtlich! */
}
```

Listing 2: Beispiel zu Regeln 8 und 9.

Wie man in Listing 2 sehen kann, ist die Struktur von zusammenhängenden `if`/`else`-Folgen der Struktur einer `switch`-Anweisung ähnlich. Doch während sich die Blöcke der Fälle 1 bis 4 in Listing 2 gegenseitig ausschließen, erlaubt die `switch`-Anweisung durch das gezielte Weglassen der Instruktion `break` auch mehrere Optionen zu durchlaufen. Diese Technik erlaubt die Erzeugung sehr effizienten Codes, führt aber leicht zu Fehlinterpretationen. Daher sollte man diesen Trick immer dokumentieren, wie Listing 3 zeigt. Im Sinne defensiver Programmierung sollte auch immer ein `default`-Zweig vorhanden sein.

Regel 10: Struktur der switch-Anweisung

Eine `switch`-Anweisung muss die in Listing 3 gezeigte Form haben. Im speziellen muss das Auslassen der `break`-Instruktion dokumentiert sein und ein `default`-Zweig vorhanden sein.

```
switch (iExpression)
{
   case OPTION1:
      /* Code zu diesem Fall.
       * Nach dem Fall eine Leerzeile. */
      break;

   case OPTION2:
   case OPTION3:
   case OPTION4:
      /* Code zu mehreren Fällen, Leerzeile. */
      break;

   case OPTION5:
      /* Code zu Fall 5 und ein verpflichtender Kommentar
       * am Ende, weil es hier absichtlich kein "break"
       * gibt. */
      /* ACHTUNG ---> KEIN break */

   case OPTION6:
      /* letzter Fall, ggf. letzter gültiger Fall überhaupt */
      break;

   default:
      FehlerReport(__LINE__);
      break;
} /* switch */
```

Listing 3: Gut gewählte Struktur einer switch/break-Anweisung.

Die `break`-Instruktion begegnet uns auch in Schleifen und ermöglicht einen Abbruch der Schleife. Speziell in verschachtelten Routinen sollte man bei der Instruktion kommentieren, was eigentlich abgebrochen wird. Das gleiche gilt für die Anweisung `continue`.

Regel 11: Kommentare für break/continue

Zu jeder `break`- oder `continue`-Instruktion in einer Schleife muss ein Kommentar angeben, welche Schleife betroffen ist.

Listing 4 illustriert eindrucksvoll wie Kommentare die Verschachtelungen verdeutlichen können und wie wichtig es daher ist, die Sprungbefehle `break` und `continue` mit Kommentaren zu versehen. Ein weiterer Befehl erlaubt uns vorzeitig aus dem Programmfluss auszubrechen: `return`. Strikte Codierungsstandards für C gestatten nur exakt eine `return`-Anweisung pro Funktion. Die Lesbarkeit von Code kann aber aufgrund dieser Restriktion sehr leiden, besonders wenn eine hohe Verschachtelungstiefe von `if`-Instruktionen verschiedene Fehlerfälle unterscheidet. Auch die korrekte Freigabe lokal benutzter Ressourcen kann schwieriger werden. Ein Trick dem beizukommen ist eine Dummy-Schleife, wie sie in Listing 5 zu sehen ist, mit definierten Variablenwerten für nicht benutzte Ressourcen zu kombinieren. Wir sehen in Listing 5, dass das Blockende der Schleife eigentlich als Sprungmarke verwendet wird und die Schleifenabbrüche dort die Funktion einer `goto`-Anweisung erfüllen.

```
if (bBlablabla)
{
   /* ... */
   for (i = 0; i < 20; i++)
   {
      while (bIrgenwas)
      {
         /* ... */
         if (bNochwas)
         {
            break;   /* Ausstieg aus while */
         }
         /* ... */
      }
      if (bIrgendwas && bNochwas)
      {
         continue;   /* weiter mit i-Schleife */
      }
   } /* for i */
} /* if bBlablabla */
```

Listing 4: Beispiel zur Wichtigkeit von Kommentaren bei break und continue.

Die `goto`-Anweisung ist nicht gerade populär in strukturierten Programmiersprachen und daher verbieten sie viele Codierungsstandards. Für Eingebettete Systeme muss aber gelegentlich auf elegantes Design verzichtet werden, wenn effiziente Code-Generierung im Vordergrund steht. Wenn in Listing 5 die `break`-Instruktionen durch `goto`-Anweisungen ersetzt werden, dann kann man auch nicht plötzlich von schlechtem Design sprechen, nur weil von `goto` Gebrauch gemacht wird. Trotzdem empfehlen wir

Empfehlung 12: Goto-Anweisung

Die `goto`-Instruktion sollte nicht verwendet werden.

weil `goto` unglaublich gefährliche Konstrukte erlaubt, wie Listing 6 demonstriert. Listing 6 hat noch mit zwei weiteren unglaublichen Stilblüten aufzuwarten. Im Initialisierungsteil der `for`-Schleife ist ein Funktionsaufruf zu finden, der überhaupt nichts mit der Initialisierung des Schleifenzählers zu tun hat; der Schleifenzähler ist als Index von `piArray[]` verwendet und würde einen ungültigen Index erreichen, weil in der Abbruchbedingung der Schleife ein Größer/gleich- statt einem Größer-Zeichen verwendet wird. Daher

Regel 13: Verwendung des Schleifenzählers in der for-Schleife

Die Ausdrücke in den runden Klammer der `for`-Instruktion dürfen nur zum Setzen, Prüfen und Verändern des Schleifenzählers dienen.

Empfehlung 14: Abbruchbedingung in der for-Schleife

Die Abbruchbedingung des Schleifenzählers sollte mit einem Größer- oder Kleiner-Zeichen erfolgen, nicht mit größer/gleich oder kleiner/gleich.

```
void Funktion(void)
{
   /* alle verwendeten Resourcen müssen im Kontext der
    * Funktion definiert werden. */
   TResource1 *ptRes1 = NULL;
   TResource2 *ptRes2 = NULL;
   TResource3 *ptRes3 = NULL;

   do /* Dummy-Schleife */
   {
      ptRes1 = malloc(sizeof(TResource1));
      if (NULL == ptRes1)
      {
         FehlerReport(KEIN_SPEICHER_FUER_RES1);
         break; /* Ausstieg aus Funktion(). */
      }

      if (...)
      {
         ptRes2 = malloc(sizeof(Tresource2));
         if (NULL == ptRes2)
         {
            FehlerReport(KEIN_SPEICHER_FUER_RES2);
            break; /* Ausstieg aus Funktion(). */
         }

      }

      ptRes3 = malloc(sizeof(Tresource2));
      if (NULL == ptRes3)
      {
         FehlerReport(KEIN_SPEICHER_FUER_RES3);
         break; /* Ausstieg aus Funktion(). */
      }

      /* auch die folgenden Zeilen müssen nicht in einem
       * else/else/else-Block stehen, der alle Fehlerfälle
       * ausschließt sondern stehen einfach auf gleicher
       * Einrückungstiefe wie alle anderen Instruktionen
       * der Funktion die sich nicht um das Bereitstellen
       * oder Freigeben von Ressourcen drehen. */
      DoSomething();
      DoAnything();

   } while (0);  /* Dummy-Schleife */

   /* die gleichen Aufräumarbeiten für alle Fehlerfälle
    * und die fehlerfreie Ausführung der Funktion: */
   if (ptRes1 != NULL) free(ptRes1);
   if (ptRes2 != NULL) free(ptRes2);
   if (ptRes3 != NULL) free(ptRes3);

   return;          /* das verpflichtende EINZIGE return */
} /* Funktion() */
```

Listing 5: Eine Dummy-Schleife und definierte Werte (`NULL`) für nicht benutzte Resourcen erlauben die Verwendung von einem einzigen Exit-Pfad pro Funktion ohne die Verschachtelungstiefe unnötig in die Höhe zu treiben.

Vorschriften an das Design

In jedem Grundkurs zu Softwaredesign wird man aufgeklärt, dass globale Variablen schlecht sind und dass man möglichst wenige Schnittstellen zwischen Programm-Modulen (C-Quelldateien) haben sollte. Als Begründung wird meist Übersichtlichkeit und Wartbarkeit des Programms angegeben. Nun hat man bei der Programmierung Eingebetteter Systeme oft ganz andere Sorgen als ein elegantes Design: Code muss sehr schnell sein und wenig Speicher verbrauchen. Daher werden die einmal gelernten Design-Richtlinien gerne vergessen und es wird in wildesten Stilen programmiert.

Davon ist dringend abzuraten. Eingebettete Systeme haben nämlich noch eine Eigenschaft, die vom Projektmanagement gerne unter den Tisch fallen gelassen wird: sie sind nicht einfach zu testen. Je sauberer und modularer das Design ist, desto besser lassen sich Programmteile isolieren und testen. Auch der Umfang von Software/Software-Integrationstests wird durch ein durchdachtes Design drastisch reduziert [5]. Daher folgende dringende Empfehlungen:

Empfehlung 15: Modulares Design

> Die C-Software sollte modular und bestenfalls objektorientiert strukturiert werden. Es sollte möglichst wenige Schnittstellen (Programmaufrufe, gemeinsame Variablen) zwischen den Quelldateien geben.

Empfehlung 16: Design und Effizienz

> Software sollte zuerst nach den Anforderungen guten Designs entworfen werden erst optimiert werden, wenn es sein muss.

Sauber verknüpfte Quelldateien lassen sich leicht neu arrangieren und wiederverwenden. Es ist daher ratsam immer alle notwendigen Deklarationen zu inkludieren, sodass jede Datei ohne Vorbedingungen separat übersetzt werden kann:

Empfehlung 17: Reihenfolge und #include

> Jede Quelldatei sollte einzeln übersetzt werden können. Die Reihenfolge der #include-Anweisungen in einer Quelldatei sollte keine Rolle spielen.

Und wenn Sie die Verzeichnisstruktur ändern müssen oder ein schlechtes Programm zum Configuration Management haben wird die Befolgung folgender Empfehlung viel Ärger ersparen.

Empfehlung 18: Dateinamen

> Jeder Dateiname sollte im Projekt nur einmal vorkommen.

Auch in Details des Designs kann durch Beachtung von Regeln die Zusammenarbeit im Team vereinfacht werden. C ist keine objektorientierte Sprache und Funktionen müssen als Parameter ggf. eine Referenz auf ein betroffenes Objekt und ein anwendbarer Parameter übergeben werden. Beispielsweise bei

```
void SetPort(int PortID, int iValue);
```

Die hier gewählte Reihenfolge der Parameter entspricht dem Prinzip zuerst das betroffene Objekt anzugeben, dann die Parameter für dieses Objekt. Dies ist das meist verwendete Prinzip. Wenn sich alle Mitglieder eines Teams daran halten, dann gibt es keine Verwirrungen, weil niemand auf die Idee kommt

```
void SetPort(int iValue, int PortID);
```

zu definieren. Also lohnt es sich hier eine Regel zu definieren. Etwa indem die Reihenfolge der Parameter durch die Reihenfolge der im Bezeichner der Funktion vorkommenden Hauptwörter definiert wird, wie in

```
WriteSampleToDAC(int iSample, int iConverterID);
```

oder indem immer zuerst das betroffene Objekt stehen muss, was am besten mit einem objektorientierten Design vereinbar ist und auch von allen mir bekannten C-Code-Generatoren so gemacht wird. Leider ist die Sprache C hier selbst inkonsistent wie man an den Funktionen `fputc()` und `fprintf()` erkennt.

Ein weiteres Design-Detail kann durch Standardisierung Reibungsflächen bei der Zusammenarbeit im Team glätten. Ich empfehle wärmstens die Reihenfolge der Deklarationen in Dateien zu reglementieren. Beispielsweise:

Regel 19: Reihenfolge von Deklarationen in Header-Dateien

In einer Header-Datei müssen Deklarationen in folgender Reihenfolge erfolgen: Makros, Typen, Variablen, Initialisierungsroutinen, andere Funktionen.

Regel 20: Reihenfolge von Definitionen und Deklarationen

In einer Implementierungs-Datei müssen Deklarationen und Definitionen in folgender Reihenfolge erfolgen: verpflichtendes `#include` der Header-Datei, Makros, Typen, globale Variable, lokale Variable (das sind Variable mit dem Attribut `static`), Prototypen von lokalen Funktionen, globale Funktionen, lokale Funktionen.

Undefinierte Ergebnisse und Schnittstellen

Die Sprache C lässt den Compilerherstellern einige Freiheiten. Beispielsweise ist die Reihenfolge der Auswertung von arithmetischen Ausdrücken nicht definiert, sofern die Mathematik offen lässt welcher Teilausdruck zuerst errechnet werden muss. Des weiteren ist das Ergebnis eines Right-Shift von negativen ganzen Zahlen nicht definiert. Ein Compiler kann wahlweise das neue höherwertige Bit mit einer Eins oder einer Null auffüllen. Zu beiden genannten Fehlerquellen finden Sie in Listing 6 in der Funktion `Horror()` ein Beispiel. Solche Ambivalenzen sind vielen C-Profis bekannt, sollten aber durch Regeln in einem C-Codierungsstandard erfasst werden um Junior-Programmierer zu warnen.

Beachten Sie auch, dass in C beim Prototyp einer Funktion die Argumente undefiniert sind, wenn keine angegeben werden, so wie in Listing 6 zu sehen ist. Dies kann natürlich zu Problemen bei der Integration von verschiedenen Modulen führen.

Regel 21: Prototypen

Zu jeder Funktion muss ein Prototyp geschrieben werden der alle Argumente definiert.

Wenn wir Empfehlung 15 folgen, dann ist die Header-Datei der richtige Platz für Prototypen global sichtbarer Funktionen und die Implementierungs-Datei der Platz von Prototypen lokal sichtbarer Funktion, wie in Regel 20 bereits festgehalten. Gute Lint-Programme überprüfen die Einhaltung von Regel 21 automatisch.

Wenn ein Compiler Inline-Funktionen zur Verfügung stellt und Portierbarkeit des Codes nicht gefordert ist, dann empfehle ich die Verwendung von Inline-Funktion der Verwendung von funktionsartigen Makros vorzuziehen, auch wenn es sich nicht um ANSI-C handelt. Beim Aufruf einer Funktion prüft der Compiler Typen und Seiteneffekte sind unwahrscheinlicher als beim Makro. Beispiele, wo eine Inline-Funktion Fehler vermieden hätte, die bei der Verwendung von funktionsartigen Makros passieren können, sind in der Funktion `Horror()` in Listing 6 zu finden.

Die letzte Warnung zum Thema undefinierte Ergebnisse gilt Puffergrößen. Das ungeprüfte Überschreiten von Feldgrenzen ist ein schwer zu erkennender Fehler und die von Hackern am meisten benutzte Sicherheitslücke in vernetzten Systemen. In C gibt es keine Überprüfung von Array-Indizes und daher kann man dieses Problem nicht automatisch unterbinden. Was aber im Rahmen eines Codierungsstandards möglich ist, ist die Verwendung von Funktionen zu verbieten, die Feldgrenzen nicht überprüfen. Ein gutes Beispiel dazu ist `sprintf()`: ist der erzeugte String zu groß für den verwendeten Puffer, so ist alles möglich, vom glimpflichen Verlauf bis zum Programmabsturz an einer völlig anderen Stelle. Die meisten Compiler bieten eine Umgehung solcher potentiellen Fehlerquellen an, in unserem Beispiel ist das die Funktion `snprintf()`.

Die Schlussbemerkung

Die Beachtung von Codierungsstandards ist eine wichtige Maßnahme zur Qualitätsverbesserung und Effizienzsteigerung in der teamorientierten Software-Entwicklung. Codierungsstandards findet man im „erzkonservativen Sektor“, der Entwicklung von Software für sicherheitskritische Systeme, genauso wie bei den „Alternativen“, den Propheten von Extreme Programming (XP) und Co. Ihre Einführung und Durchsetzung kostet aber einiges an Kraft. Wie XP-Guru Kent Beck in seinem letzten Buch bemerkt: „Programmierer sind Individualisten, sie würden eher die Firma verlassen, als ihre geschwungenen Klammern anders zu platzien ...“ [6]. Doch zum Glück gibt uns der Guru noch einen wichtigen Hinweis: „... es sei denn man kann sie davon überzeugen, dass die Chance Mitglied im Gewinner-Team zu sein dann größer ist.“

Referenzen

[1] Codierungsstandards in der Software-Entwicklung für Eingebettete Systeme.
[2] Ein guter Programmstil will standardisiert sein.
[3] Professionelles Codelayout.
[4] Steve Dewhurst: One at a Time, Please. *C/C++ Users Journal* 2001, H. 8, S. 46 bis 50.
[5] European Space Agency, ESA-PSS-05-10: Guide to Software Verification and Validation. Siehe zum Beispiel http://esapub.esrin.esa.it/pss/pss-ct03.htm.
[6] Kent Beck: Extreme Programming Explained. Addison-Wesley, 2001.

```
#include <stdio.h>

#define SQUARE1(x) (x*x)      /* schlechte definition */
#define SQUARE2(x) ((x)*(x))  /* besser, aber nicht perfekt */

void Horror();          /* schlechter Prototyp von Horror() */

void Horror(int iEgal)
{
      short int x = 1;
      short int y = x / ++x;    /* y ist entweder 0 oder 1 */
      x = -12;
      x >>= 1;             /* x ist entweder -6 oder 32762 */
      x = SQUARE1(3 + 3);  /* x ist nicht 36 */
      x = 2;
      x = SQUARE2(x++);    /* x ist nicht 4 */
}

/************************************************************/
void main()
/*
 * Dieses Programm wird fehlerfrei und ohne Warnungen unter
 * MS Visual C++ 6.0 übersetzt.
 ************************************************************/
{
   int i = 12;
   int piArray[10];

   goto InDieSchleife;
   for (i = 0, Horror(1); i <= 10; i++)
   {
      printf("%d, ", i);
      piArray[i] = 0;   /* Schleifenzähler schlecht gewählt! */
InDieSchleife:
      printf("Schleife wird nie initialisiert.\n"
             "Horror() wird nicht aufgerufen\n" , i);
   }

   if (i > 12)
   {
      printf("Quer durch ");
      goto AndererProgrammpfad;
   }
   else
   {
AndererProgrammpfad:
      printf("Programmpfade\n");
   }

   goto MitteVonBlock;
   {
      int k;  /* Initialisierung hier hätte jeder Compiler verweigert */
      k = 9;  /* Umgehung: Zuweisung statt Initialisierung */
MitteVonBlock:
      printf("k: %d\n", k);  /* Wert von k nicht definiert */
   }
}
```

Listing 6: Gute Gründe `goto` aus Ihren Programmen zu verbannen und Beispiele für undefinierte Ergebnisse.

Work Instruction Risk based Testing

Introduction

The idea of a risk based test strategy is to distribute test effort dependent on risk estimation. The available test capacity is distributed in a reasonable way. More time is spent on testing more risky program parts, less time is spent on testing less risky parts of the software.

Responsibilities

The risk based test strategy has to be planned by the project software responsible or a person who is assigned by him. For risk estimation all necessary stakeholders (system developers, software developers, customers, testers, etc.) may be involved.

Work Instruction

The software has to be split in a manageable number of units. The units must be distinguishable by risk estimation. A resonable guideline for the size of the units is the following: The units should differ in risk likelihood attributes and the units should differ in risk impact attributes. Examples for units are source code modules, functional components, or library packages. The splitting should be stopped when no existing unit contains parts that differ considerablly in a likelihood or in an impact attribute.

Likelihood Attributes

For each unit determine a scale factor as the largest value of any of the qualities described in the following table.

Scale	Size/Complexity	Number of Former Changes	Level of Optimization (Resources)	Defects Found Earlier
1	very small	established functionality, never changed	no optimization	never
3	small	infrequently revised	less optimization	infrequently
6	large	frequently revised	heavy optimization	frequently
9	very large	permanently revised	very heavy optimization	permanently

Classify the software (change) you are going to make according to the following table and take the largest scale factor, as before.

Scale	Modification	Requirements + Design Quality	Newness of used Technologies
1	not modified/bugfix	clear requirements + design	established technologies used (technology is well-known and is used for this functionality)
3	small modifications	clear requirements, design partly unclear	well-known technologies used (but it was not used for this functionality)
6	essential modifications	design + requirements partly unclear	new technologies, but it is used for a while. It is sure that it is the required solution but it is not controlled completely.
9	completely new developed	design + requirements completely unclear	new technologies used for the first time

Then classify external circumstances according the the table below and note the largest scale factor corresponding to these influences.

Scale	Experience	External Development	Change of Developer	Time Pressure
1	very experienced developer(s)	no problems with external development	no change of developers	normal time pressure
3	experienced developer(s)	unimportant problems with external development	uncritical change of developers (e.g.: same know-how, change very early in development phase)	increased time pressure
6	inexperienced developer(s)	important problems with external development	very critical change of developers (e.g.: change very late in development phase)	high time pressure
9	very inexperienced developer(s)	critical external development	very critical change of developers (e.g.: change very late in development phase, many developers changed)	very high time pressure (e.g. change of requirement near delivery)

Impact Attributes

Scale	Customer Importance
1	functionality is nice to have
3	functionality is standard requirement
6	functionality is important (e.g. new functionality based on changed laws)
9	functionality is very important for the customer (e.g. a functionality which differs the customer from other competitors)

Scale	Modification
1	customer demands rework
3	customer demands rework and will not pay the whole amount
6	customer demands rework and demands a fine
9	customer demands rework and demands a very high fine

Scale	Safety
2	not safety relevant
6	product damage possible
9	impact on person's safety possible

Risk Assessment and Test Extent

Now feed the scales into the Excel sheet found on the management intranet page and press OK to compute the risk level ranging from 1 to 3. The following table gives the test procedure and tool required for the obtained risk level.

Risk Level	Test Methods	Test Tools
1	System Testing only	Simulator or target hardware
2	Unit Testing (100% decision coverage) and System Testing	Cantata++ 3.0 on host, or tgt, target hardware
3	Unit Testing (100% MC/DC), Integration Testing (100% Call/Pair Covg.), System Testing	Cantata++ 3.0 on target, Cantata++ 3.0 on target, target hardware

Warning: this plan is pure fiction. It is used in an exercise and should not be used in real life without implementing the conclusions drawn in the exercise.

REVIEW ITEM DISCREPANCY RID NO

DATE

ORIGINATOR

1. DOCUMENT TITLE:

2. DOCUMENT REFERENCE NUMBER:

3. DOCUMENT ISSUE/REVISION NUMBER:

4. PROBLEM LOCATION:

5. PROBLEM DESCRIPTION:

6. RECOMMENDED SOLUTION;

7. AUTHOR S RESPONSE:

8. REVIEW DECISION: CLOSE/UPDATE/ACTION/REJECT (underline choice)

Annotated References

Standards, online literature, magazines, and books.

[Bach99] James Bach: Test Automation Snake Oil. See http://www.jamesbach.com; reprinted from *Windows Tech Journal* 10/96.

Funny to read article. Teaches us that test automation is good but should be implemented with care.

[Beck01] Kent Beck: *Extreme Programming Explained.* Addison-Wesley, 2001.

Kent Beck is one of the protagonists on the XP-wave. His book is easy to read and very American: XP is cool, funny, and you‘ll make a lot of money.

[Fagan86] M.E. Fagan: Advances in Software Inspections. *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 7, July 1986.

You don’t want to read that. I haven’t read it either.

[FDA02] U.S. Department Of Health and Human Services, Food and Drug Administration, CDRH: *General Principles of Software Validation; Final Guidance for Industry and FDA Staff.* January 11, 2002, published on the WWW.

Unless you want to submit a medical device to the US market there is no need to read that publication. However, it is really a good collection of advices.

[Hall03] Payson Hall: Knowing the Odds. A practical, three-step risk management process you can accomplish in one afternoon. *STQE, The Software Testing & Quality Engineering Magazine.* Vol 5, issue 2, March/April 2003, pp. 34-40.

After reading articles in this magazine you will wonder how it is possible to write so many lines about so obvious things.

[Myers78] Glenford Myers: *The Art of Software Testing*. John Wiley & Sons, 1978.

A classic. However, even though software testing techniques change very slowly over time, it clearly does not reflect the state of the art.

[Perry95] William Perry: *Effective Methods for Software Testing*. John Wiley & Sons, 1995.

Don’t read that. Full of bla bla, tables, and metrics nobody ever uses.

[IE3RA] *ANSI/IEEE Std 1028-1988, IEEE Standard for Software Reviews and Audits.*

Better let it in the shelf unless you want to become an SQA or auditor.

[IEEE] *IEEE Standard Glossary of Software Engineering Terminology 1990.*

Good but quite expensive. However, you will find cites in the WWW for many of the definitions of this standard.

[IPL99] *Advanced Coverage Metrics for Object-Oriented Software.* IPL Information Processing Ltd., 1999.

This is a white paper of a company selling unit test tools. It is appended to the lecture notes.

[PSS-05-0] European Space Agency Board for Software Standardisation and Control: ESA software engineering standards. ESA PSS-05-0 Issue 2, February 1991.

A very good overview of a quite formal software lifecycle model. Can be purchased from ESA. Black copies of this standard can be found in the web.

[PSS-05-10] European Space Agency Board for Software Standardisation and Control: *Guide to software verification and validation.* ESA PSS-05-10 Issue 1, February 1994.

Gets into more detail than [PSS-05-0] with respect to software verification.

[PSS-05-03] European Space Agency Board for Software Standardisation and Control: *Guide to the software requirements definition phase.* ESA PSS-05-03 Issue 1, October 1991.

Gets into more detail than [PSS-05-0] with respect to requirements management and engineering.

[Rosenb02] Linda Rosenberg: What is Software Quality Assurance?. *STSC Crosstalk*, May 2002.

STSC Crosstalk is a free web-based publication for software quality in military applications.

[RUP] IBM, *Rational Unified Process.* See http://www.rational.com

A set of document templates and tools that support the software life cycle together with a HTML-based description of software project management issues and examples.